Universidad
Carlos III de Madrid

UNIVERSIDAD CARLOS III DE MADRID
SCHOOL OF ENGINEERING

BACHELOR'S DEGREE IN TELECOMMUNICATION TECHNOLOGIES

BACHELOR'S THESIS

EVALUATION OF MACHINE LEARNING METHODS IN WEKA

AUTHOR: **ANTONIO A. PASTOR VALLES**
TUTOR: **CARMEN PELAEZ MORENO**
CO-TUTOR: **FRANCISCO J. VALVERDE ALBACETE**

September of 2015

Acknowledgements

This project closes a stage of my life. The journey here has not followed a straight path, and that has lengthen the way. In any case, I could not have made this long trip without the aid, support, and encouragement of my family and closest friends and classmates. I will be forever grateful to all of them.

I have to thank my tutors of this work, Carmen Peláez Moreno and Francisco José Valverde Albacete, their support and always constructive feedback. Thank you for trusting in my ability to carry out the project and helping me on it. I hope you like the outcome and we can make it more complete soon.

In the last months I have been fortunate of starting new interesting projects in research. This has motivated me to close this stage for starting a new one. I take this lines to thank the opportunity that my new tutors are giving me.

Everyone knows that debugging is twice as hard as writing a program in the first place. So if you are as clever as you can be when you write it, how will you ever debug it?
Brian Kernighan

Abstract

This document presents a software plug-in to endow the Weka machine learning suite with the complete set of information-theoretic tools described by Valverde-Albacete and Peláez-Moreno [*Pattern Recognition Letters* 31.12 (2010) and *PLoS ONE* 9.1 (2014)]. The utility of these tools is more evident in multi-class classification, but they can be used as well for binary tasks.

The Entropy Triangle is an exploratory analysis method that we implemented as an interactive visualization plugin for Weka. The Entropy Triangle represents in a De Finetti diagram, or ternary plot, a balance equation of entropies for the estimated distributions of the input and the output of classifiers. This diagram provides, at a glance, complete information of the confusion matrix in terms of information theory.

Besides the Entropy Triangle, we implement in the package some useful metrics for the assessment of classifiers based on the perplexity. In the context of classification, the perplexity represents the effective number of classes for the classification task, which makes it a useful measure of the propagation of information. Among these metrics, we highlight the Entropy Modified Accuracy, recommended to rank classifiers, and the Normalized Information Transfer factor, to measure the classifiers level of understanding of the underlying patterns of the task.

The *Waikato Environment for Knowledge Analysis* (WEKA) is a workbench for machine learning and data mining developed at the University of Waikato, New Zealand. Weka has different Graphical User Interfaces available, that let the user choose from an user friendly interactive explorer, to an automated approach where multiple experiments can be statistically compared at the same time. An important feature of Weka is the possibility to use it as a framework for the implementation of algorithms, evaluation metrics and visualization tools by means of added components.

In this document we describe the design and development of the software package. Before that, we set the theoretical backdrop reviewing the implemented tools and their mathematical background. To illustrate the software features and the utility of the tools, we present an example with a multi-class dataset in which we unbalance the class distribution in different ways. Additionally, we introduce how to use the plug-in programmatically with a guided example. Finally, we review the project in hindsight and propose future work.

Keywords: Machine Learning, Classification, Evaluation, Information-Theory, Weka.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This document presents a software plug-in to endow the Weka machine learning suite [8] with a set of information-theoretic tools for the assessment of classifiers [22, 23]. The utility of these tools is more evident in multi-class classification, but they can be used as well for binary tasks.

1.1 Motivation

In academia, there is usually a time lag between the advance of the theoretic field and the practical adoption of new techniques. Not to mention its application at the industry, where discoveries often go completely unnoticed. We refer to it as the *implementation gap*.

This gap may be due to different facts, but a common one, is the lack of a software implementation for the theoretical advances that allows other researchers or developers to get familiarized with the new techniques without investing a large amount of time.

Engineering students can help to fill this gap developing implementations of state-of-the-art techniques for their academic work. An experience with which students can:

- Practice and develop further technical and management skills.
- Gain a perspective of the research world.
- Learn new theoretical concepts on interesting topics.
- Facilitate the general access to cutting-edge research.
- Face the challenge of solving real-world engineering problems.

Open source software is widely used in academia for the implementation of research work. The public release of code agrees with the scholarly spirit, where common knowledge is above the individual's profit. It is a foundation for a collaborative environment that can address large-scale developments and projects that endure over time.

This project aims at addressing the implementation gap for recent work [22, 23] on the evaluation of supervised machine learning methods, specifically classifiers, by means of tools from information theory, with the goal of providing an implementation integrated in a well-known machine learning open source project.

1.2 Outline

This document is organized as follows. In Chapter 2 we review the theory behind the tools and previous attempts at implementing them. We next review the Weka machine learning suite. Once set the context, we identify the project objectives and requirements for the software.

In Chapter 3, we review Weka's architecture identifying the requirements both for the program to recognize the new tools and for bundling them in a plugin package. Then, we present the design to implement the software.

Next, in Chapter 4, we present some use cases to illustrate the software features and the utility of the tools. Here, we show an example of use for unbalanced datasets and we introduce how to use the plug-in programmatically, providing the code of this example.

Finally, Chapter 5 reviews the project in hindsight and proposes future work.

The first two annexes include the installation manual and user manual. In the third appendix we overview the legal framework applicable to the project, the license of the software and this document are included. The fourth and fifth appendix estimate a budget for funding this project and an outline of the planning. And finally, the closing appendix is an extended abstract of this document.

Chapter 2

Background

2.1 Information-Theoretic Evaluation of Classifiers

The tools implemented in this software package are widely explained and justified in [23] and [22]. In this section we review them and their mathematical background to provide a reference within the document.

The use of information theory to assess classifiers is grounded on the analogy between a classification process and a communication channel [16]. In this context, the input and the output of the classifier can be viewed as discrete random variables X and Y, respectively, with k—the number of classes—different possible values. Their marginal and joint probabilities can be estimated from the confusion matrix of the classifier testing.

2.1.1 Confusion matrix

The confusion matrix C_{XY} tallies the label class of the classified instances against their predicted class. We use the criterion that the rows correspond to the label class, the input, and the columns to the predicted class, the output, so that the element (i, j) of the matrix is the number of instances in the test set which, belonging to the class x_i, are classified as y_j.

Then, the maximum likelihood estimate for the joint probability of every label-prediction pair is the element of the matrix at such index divided by the total number of instances of the test set

$$\hat{P}_{XY} = C_{XY}/N$$

wherefore we can estimate the joint probabilities and their marginals.

Although the confusion matrix contains complete information to evaluate the classifier, its direct inspection, e.g. as a *heatmap*, is not practical. Specially in the multi-class classification problems, the information-theoretic tools will help us to extract the information in the confusion matrix in a more apprehensive form. This is the starting point of the entropies analysis in the next section.

2.1.2 Entropic analysis of the confusion matrix

Entropy is the basic measure of information, and measures the uncertainty in the outcome of a random experiment or variable X [21]

$$H_X = E\left[\log \frac{1}{P_X}\right] = -\sum_i p(x_i) \log(p(x_i))$$

where we have used binary logarithms. Mathematically speaking, since it is a logarithmic measure, the operations in terms of entropies are simplified to sums. Therefore, the use of entropies is more convenient for our analysis than handling the probabilities directly.

In addition, we can define more useful measures in terms of entropy. For instance, we can define the mutual information of two variables [5], or the variation of information between them [15]. This measures are important by their own and, also, will let us simplify the equations in the entropic analysis with proper naming.

Mutual Information

The mutual information of two variables measures the amount of information that these variables share. This is the information, or uncertainty, that remains when subtracting from one variable the uncertainty of that same variable given the other

$$MI_{XY} = H_X - H_{X|Y} = -\sum_{i,j} p(x_i, y_j) log_2 \frac{p(x_i, y_j)}{p(x_i)p(y_j)} \tag{2.1}$$

so that two independent variables does not have any mutual information. Notice that it could be formulated as $MI_{XY} = H_Y - H_{Y|X}$ as well.

Variation of Information

The variation of information is the opposite concept to the mutual information and can be interpreted as the distance between the two random variables [15]. The variation of information results of extracting the mutual information present in each of the variables

$$VI_{XY} = H_X + H_Y - 2MI_{XY} = H_{X|Y} + H_{Y|X} \tag{2.2}$$

Balance equation of entropies

The maximum value of a entropy correspond to the uniform distribution, where all the values are equally likely. Therefore, the upper bound for the joint entropy of two variables is the case when they both are uniform and independently distributed

$$H_{P_{XY}} \leq H_{U_X \cdot U_Y} \tag{2.3}$$

Since the result is valid also for the specific case where the two variables are mutually dependent, the difference in their joint entropy from the upper bound is

$$\Delta H_{P_X \cdot P_Y} = H_{U_X \cdot U_Y} - H_{P_X \cdot P_Y} \geq 0 \tag{2.4}$$

Recall that the joint entropy of two variables is $H_{P_{XY}} = H_{P_X} + H_{P_Y} - MI_{XY}$. Here we subtract the mutual information once because is present in both variables. This equation shows that the difference between $H_{P_X \cdot P_Y}$ and the joint entropy, $H_{P_{XY}}$, is the mutual information.

$$MI_{XY} = H_{P_X \cdot P_Y} - H_{P_{XY}} \tag{2.5}$$

Fig. 2.1 shows the information diagram of a bivariate distribution as in [39] with the addition of two outer boxes. The exterior box, is the upper bound of the entropy of Eq. (2.3), and the entropy of the added areas are the values of Eqs. (2.4) and (2.5).

We can derive a balance equation of the entropies for two variables summing all the areas in Fig. 2.1

$$H_{U_X \cdot U_Y} = \Delta H_{P_X \cdot P_Y} + 2MI_{XY} + H_{P_{X|Y}} + H_{P_{Y|X}} \tag{2.6}$$

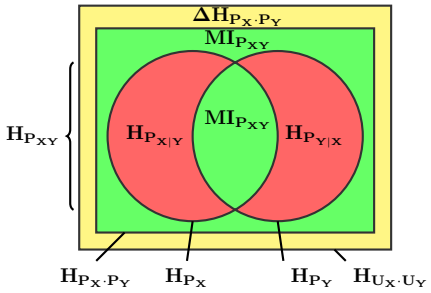


Figure 2.1: Extended information diagram of entropies for a bivariate distribution (reprinted from [22])

The last two terms correspond to the variation of information, giving an equation of three terms in the right side

$$H_{U_X \cdot U_Y} = \Delta H_{P_X \cdot P_Y} + 2MI_{XY} + VI_{XY} \tag{2.7}$$

Separate balance equations

Eq. (2.7) can be decomposed in separate equations for each variable. We take advantage that, as the entropy is a logarithmic measure based on probabilities, a product of probabilities becomes a simple addition in terms of entropy. Then, Eq. (2.4) can be expanded to

$$\Delta H_{P_X \cdot P_Y} = (H_{U_X} - H_{P_X}) + (H_{U_Y} - H_{P_Y}) = \Delta H_{P_X} + \Delta H_{P_Y} \tag{2.8}$$

We proceed alike with $H_{U_X \cdot U_Y} = H_{U_X} + H_{U_Y}$, and substitute them in Eq. (2.6). Then, we can decompose the entropy equation balance in two equations

$$H_{U_X} = \Delta H_{P_X} + MI_{XY} + H_{P_{X|Y}} \tag{2.9a}$$

$$H_{U_Y} = \Delta H_{P_Y} + MI_{XY} + H_{P_{Y|X}} \tag{2.9b}$$

Fig. 2.2 presents this equations in a modified information diagram.

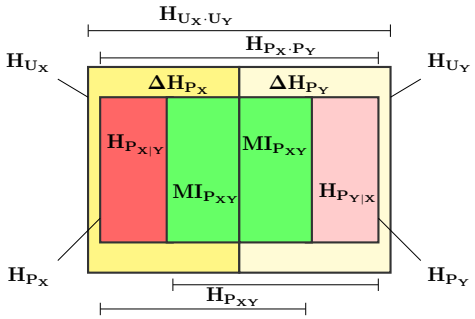


Figure 2.2: Extended information diagram of entropies for a bivariate distribution (reprinted from [22])

2.1.3 The Entropy Triangle

The Entropy Triangle represents in a De Finetti diagram, or ternary plot, the balance equation of entropies from Eq. (2.7), normalized by $H_{U_X \cdot U_Y}$

$$1 = \Delta H'_{P_X \cdot P_Y} + 2MI'_{XY} + VI'_{XY} \tag{2.10}$$

This diagram provides, at a glance, complete information of the confusion matrix in terms of information theory. The multi-class task, lacked of good visual characterizations, like the *Receiver Operating Characteristic* (ROC) curve [6] for binary classification. For this reason, the Entropy Triangle appears as a powerful tool for the reliable assessment of multi-class classifiers [23].

As an illustration of the information that gives us the different areas of the Entropy Triangle we review the borderline cases. Classifiers are drawn at sides of the triangle when one of the elements in Eq. (2.10) is zero:

Left side ($\Delta H'_{P_X \cdot P_Y} = 0$) A classifier in this side has uniform distribution for its input and output variables. Uniform balancing in the output distribution means that a classifier does not privilege any class in detriment of another.

Lower side ($MI'_{XY} = 0$) When there is no transfer of information to the output, Y is independent of X. In this case, the classifier acts randomly.

Right side ($VI'_{XY} = 0$) If there is no variation of information, there is no residual information on the conditional entropies. This means that the classifier transfers all the information available to the output.

Fig. 2.3 shows the Entropy Triangle identifying the performance qualities for classifiers located at the vertices and sides.

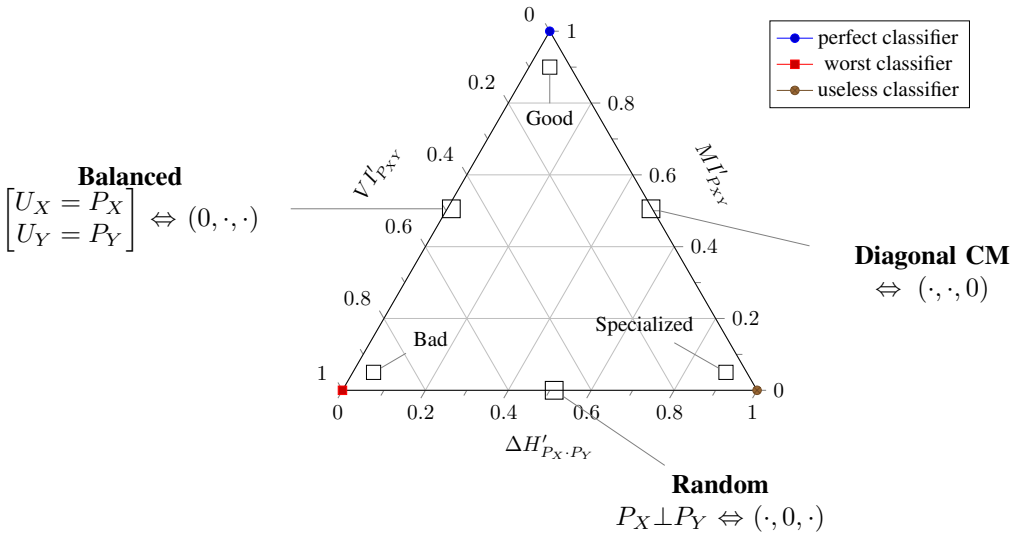


Figure 2.3: Entropy Triangle showing interpretable zones and extreme cases of classifiers (reprinted from [22])

Split Entropy Triangle

The Entropy Triangle can represent the separate balance equations of the marginal distributions (Eq. (2.9)). The equations are now normalized by H_{U_X} or H_{U_Y}, respectively

$$1 = \Delta H'_{P_X} + MI'_{XY} + VI'_X \tag{2.11a}$$

$$1 = \Delta H'_{P_Y} + MI'_{XY} + VI'_Y \tag{2.11b}$$

where $VI'_X = H'_{P_{X|Y}}$ and $VI'_Y = H'_{P_{Y|X}}$.

When the number of classes in X and Y is the same $H_{U_X} = H_{U_Y}$, the value of MI'_{XY} for the two marginals and the joint equation coincide as well.

Upper bound to the mutual information due to dataset likelihoods

From what we have seen so far, a desirable feature for a classifier is to maximize the mutual information between the input and the output. But, the mutual information is conditioned by the equation balance.

To illustrate, suppose a perfect classifier that transfers all the information it has to the output, leaving no $H_{P_{X|Y}} = 0$. Then, Eq. (2.11a) becomes $1 = \Delta H'_{P_X} + MI'_{XY}$. As P_X does not depend of the classifier, this is an upper bound for the mutual information

$$MI'_{XY} \leq 1 - \Delta H'_{P_X} \tag{2.12}$$

We can draw a line in the Entropy Triangle for $\Delta H'_{P_X}$, the normalized distance in entropy of the dataset likelihood from the maximum. This line draws an intuition of the limit of performance of the classifier due the test set.

2.1.4 Perplexity-based metrics for classifiers

The perplexity is an information-theoretic measure of difficulty used to indicate the number of possible values for discrete variables [1]. In our context, the perplexity represents the effective number of classes for the classification task, which makes it a useful measure of the propagation of information [22].

The perplexity PP of a discrete variable is calculated in terms of its entropy

$$PP_X = 2^{H_X}$$

The upper bound for the perplexity is produced for the uniform distribution. If all the classes are equally likely, the effective and the real number of classes are the same.

$$PP_{U_X} = 2^{H_{U_X}} = k \tag{2.13}$$

And the minimum value is 1 for a certain choice.

Class perplexity

We can call class perplexity to the number of effective classes due the likelihood of the input distribution

$$PP_{P_X} = 2^{H_{P_X}} = k_X \tag{2.14}$$

where $k \geq k_X \geq 1$.

Remaining perplexity

Since $H_{P_{X|Y}}$ measures the information that the classifier failed to learn, its perplexity represents the residual difficulty for the classification task after the learning process. We can call remaining perplexity to

$$PP_{P_{X|Y}} = 2^{H_{P_{X|Y}}} = k_{X|Y} \tag{2.15}$$

The information transfer factor

The information transfer factor measures the decrement in difficulty of the task as the perplexity of the mutual information between the input and the output distributions

$$\mu_{XY} = 2^{MI_{XY}} \tag{2.16}$$

Now, we can see the relation of Eq. (2.1) in terms of perplexities

$$\mu_{XY} = 2^{MI_{XY}} = 2^{H_{P_X} - H_{P_{X|Y}}} = \frac{k_X}{k_{X|Y}} \tag{2.17}$$

The higher is the information transfer factor, the easier is the task for the classifier. This is an important measure of how well did the classifier understood the underlying patterns of the data.

Entropy Modified Acurracy

The accuracy is often used to measure the classifiers performance. But, studies have demonstrated its pitfalls [2, 22]. From the relation of the accuracy with the perplexity, the entropy modified accuracy is specified in terms of perplexity as a pessimistic estimate of the accuracy [22].

The entropy modified accuracy is defined as the inverse of the remaining perplexity

$$EMA = \frac{1}{PP_{P_{X|Y}}} = \frac{1}{2^{H_{X|Y}}} = \frac{1}{k_{X|Y}} \tag{2.18}$$

where $1/k_X \leq EMA \leq 1$.

Recall that for a perfect classifier $PP_{P_{X|Y}} = 1$, since the task would not represent any difficulty. And therefore, $EMA = a(P_{XY}) = 1$, where $a(P_{XY})$ is the accuracy. The lower bound of the EMA is the class perplexity, and it is achieved when the classifier does not capture any information, $\mu_{XY} = 1$.

NIT factor

The Normalized Information Transfer factor is defined, as its name suggests, like the information transfer factor, normalized by the number of classes [22]

$$NITfactor = \frac{\mu_{XY}}{k} \tag{2.19}$$

Notice that k, the number of classes, is also the maximum perplexity possible. This means that the information transfer factor is normalized by its maximum possible value, and therefore $NITfactor \leq 1$. The minimum information transfer factor, $\mu_{XY} = 1$, gives the lower bound $1/k \leq NITfactor$.

With Eq. (2.17), we can relate the NIT factor to the EMA

$$NITfactor = \frac{\mu_{XY}}{k} = \frac{k_X}{k}\frac{1}{k_{X|Y}} = \frac{k_x}{k}\, EMA \tag{2.20}$$

where $k/k_X = 2^{H_{U_X} - H_{P_X}} = 2^{\Delta H_{P_X}}$, from Eq. (2.8) in terms of perplexities.

The relation of Eq. (2.20) between the EMA and the NIT factor also bind their bounds

$$\frac{1}{k} \leq NITfactor \leq EMA \leq 1$$

Where the EMA only can equal the lower bound $1/k$ and the NIT factor the upper bound 1 when the likelihoods are uniform. This limitations do not suppose a practical problem, but suggest different uses for both metrics. In [22], the EMA is recommended to rank classifiers, and the NIT factor to measure the classifiers level of understanding of the underlying patterns of the task.

2.2 Related work on the Entropy Triangle

For references, in [23] and [22] a Matlab package was used to explore the Entropy Triangle, NIT and EMA[1]. But this was always meant as a proof-of-concept and not for production[2]. It lacks flexibility and can only be integrated as a script into Matlab's scripting system.

An R package is being built around the Entropy Triangle[3], but it is not yet available, and the philosophy of R caters to researchers but not the casual user, hence this package does not overlap with our aim of making the technique known to a wider audience and to lower the technological barrier for using it.

2.3 The Weka software

The *Waikato Environment for Knowledge Analysis* (WEKA) [8] is a workbench for machine learning and data mining developed at the University of Waikato, New Zealand[4]. The project has more than twenty years of history [9].

Figure 2.4: The Weka logo (reprinted from [26])

Weka has large acceptance in academia and the data mining community [18]. The program appears among the most used in data mining polls [18], and is used by universities in their classes. Moreover, it has many related projects [35], and third-party components [36].

Weka has different Graphical User Interfaces (GUIs) available, that let the user choose from an user friendly interactive explorer, to an automated approach where multiple experiments can be statistically compared at the same time. Advanced users can use the program from the command line or *programmatically* in code, a mechanism available for Java, C, Python, etc. Accordingly, it includes an extensive list of algorithms, data processing tools, and visualization facilities. An important feature of Weka is the possibility to use it as a framework for the implementation of algorithms, evaluation metrics and visualization tools by means of added components. The current version of Weka (3.7) has a package manager that eases installation and sharing of custom components as *plug-in packages*.

The project documentation is extensive[5]. The main reference is an introductory level data mining book [38], currently on its third edition [17]. Also, there are two on-line courses[6], that cover some of the book topics, a

[1] `http://www.mathworks.com/matlabcentral/fileexchange/30914-entropy-triangle`
[2] Private communication with the authors.
[3] Private communication with the authors.
[4] As a curiosity, we shall mention that the program is named after a flightless bird native of New Zealand, similar to the kiwi. The bird stars in the program logo (Fig. 2.4).
[5] `http://www.cs.waikato.ac.nz/~ml/weka/documentation.html`
[6] `http://weka.waikato.ac.nz/`

complete wiki, and a maintained public mailing list. In addition, the program is rich in built-in help features and has a comprehensive manual.

The technical documentation of Weka is really helpful. The Weka API [25] is well explained and self-contained. The Weka Wiki [28] illustrates the package creation process and provides templates for the common files. And the active mailing-list gives support for unexpected issues. Not to mention that Weka is open source, and the code availability is always a good help to understand the software in the process of interfacing it.

The Weka features listed above, are the main facts that motivated our choice of Weka as the platform to implement the tools of our interest. Especially, the plug-in scheme through installable packages is essential to encourage Weka users to try new tools.

The program is implemented in Java, so the Java object-oriented architecture is a convenient environment to develop our tool packages. In addition, since we have to implement a visualization tool, the Java built-in graphic libraries are of great help. For example, making the graph elements interactive.

2.4 Objectives and Requirements

The main goal of this project is to successfully implement the information-theoretic evaluation metrics described in [22] and the Entropy Triangle, the visualization tool or *plot* introduced in [23], as a plug-in for the Weka architecture.

The produced software must meet the following functional requirements:

1. *The plug-in must endow Weka with the information-theoretic evaluation metrics described in [22].*
   (a) *The evaluation metrics must be available in all of the interfaces of Weka.*
   (b) *The evaluation metrics must integrate naturally on evaluation reports.*
2. *The plug-in must endow Weka with the Entropy Triangle introduced in [23].*
   (a) *The Entropy Triangle visualization must support interactivity.*
   It would be desirable to integrate the Entropy Triangle on all the Weka interfaces. But, since the Weka built-in visualizations are only available on the Explorer and the Weka startup window, we have relaxed this aim to make the plot as handy as possible.
   (b) *The Entropy Triangle plot must be printable/storable.*
   Another important feature for a plot, is that it can be saved. We must explore the options for that.

The package should also meet the following quality requirements:

1. *From the user's perspective,*
   (a) *The user has to feel comfortable with the plug-in installed in Weka.*
   If the plug-in is intrusive, it is more likely to be uninstalled, and therefore, the tools not used. Despite this, it has to be accessible.
   (b) *The fact that the new metrics do not belong to the Weka core should be transparent for the final user.*
   (c) *For the same reason, the appearance should be consistent with the Weka look and feel.*
2. *From a software developer perspective,*
   (a) *The package should be lightweight.*

(b) *The program should exploit the Java capabilities for dynamically loading data.*

One of the main features of Weka Explorer and visualizations tools is that they are loaded dynamically: a Weka user expects this feature seamlessly, whether the visualization is built-in or a plug-in.

(c) *Functionality already implemented in Weka—accessible via the API—should be used when possible.*

For functions not present in Weka the Java standard libraries should be used. If these options do not suit our needs on a particular feature we will consider a external library.

3. *The code should be uploaded to a server for open source software.*

Equally important to developing the software, is making it available. This way, we hope to extend its impact and get some feedback that can help to improve it.

4. *We should develop an informal web site to introduce the package features and options.*

In addition to finding the software, new users should have documentation to guide their first steps. An overview of the tools' functionality may help users to appreciate if they are interested in learning further.

Chapter 3

System Architecture and Design

3.1 System Architecture

The design of a software plug-in is constrained by the application for which it is developed. Before making any design choice, we need to analyze the Weka requirements to add visualization plug-ins and metrics for evaluating classifiers. Likewise, we need to know how to bundle our software for the Weka package manager [32].

In this section we give a brief description of the Weka interfacing features used in the plug-in. The specifications to extend Weka are explained in the Weka manual [3] and in the official wiki [28]. For more details, the complete reference of Weka classes and methods can be found in the Weka API (Application Programming Interface) [25].

Weka has an object oriented architecture and is implemented in the Java programming language. As it is natural in Java, Weka uses inheritance to extend functionality and interfaces to interact with plug-ins.

3.1.1 Weka version for the plug-in

Weka follows the Linux model of releases [27], it means that two branches of the program are maintained: stable and developmental. As its name suggests, the development version has more features than the stable one, although this functionality may suffer from changes until ported to a stable release. On the other hand, developing our package for the stable version would provide stability in the compatibility with future releases of Weka.

However, the pluggable evaluation metrics [34] and the package manager [33] are relatively recent features in Weka, only present in the current development version (3.7). Therefore, we decided to develop our package for the development version. Hence, we will have to check the compatibility of our package on new Weka releases and update it when necessary.

As of August 2015, the latest development release of Weka is 3.7.12. Therefore, this is the version for which we designed the package and the one referenced in this document.

3.1.2 Pluggable evaluation metrics

The plug-in manager handles the addition of new evaluation metrics to Weka, and makes them available in all the Weka user interfaces.

Plugin Manager configuration file

The metrics of the plug-in have to be listed in a configuration file at the package root directory named `PluginManager.props` [32]. All the evaluation metrics included in the package must be indicated in the same property statement, in a comma separated list. In case that metrics are added in multiple statements, the plug-in manager will retain only the last one. The trick to get a more readable format is to break the statement in multiple lines placing a backslash (\ ) before every line break. Table 3.1 shows the `PluginManager.props` file.

```
weka.classifiers.evaluation.AbstractEvaluationMetric = \
    weka.etplugin.metrics.Ema, \
    weka.etplugin.metrics.Nit, \
    weka.etplugin.metrics.ClassPerplexity, \
    weka.etplugin.metrics.RemanentPerplexity, \
    weka.etplugin.metrics.InformationTransferFactor, \
    weka.etplugin.metrics.VariationOfInformation
```

Table 3.1: `PluginManager.props` configuration file (Adapted from [32])

Classes and interfaces

The metrics classes must extend the `weka.classifiers.evaluation.AbstractEvaluationMetric` abstract class and implement one of the available interfaces, attending to the type of metric.

For the metrics of our package we use the following two interfaces:

- `weka.classifiers.evaluation.StandardEvaluationMetric`

  The metrics implementing this interface will be printed by default in the evaluation report. We will use this interface for the Entropy Modulated Accuracy (EMA) and the Normalized Information Transfer factor (NIT) (see section 2.1) because, we assume that if the user installs the plug-in, both metrics are likely to be of his interest. Nonetheless, the user can remove them from the report via the Weka options.

- `weka.classifiers.evaluation.InformationTheoreticEvaluationMetric`

  This interface is intended for information theoretic metrics, disabled by default. The user can activate them in the options. As the Entropy Triangle plot is a better place to evaluate the information provided by the package metrics, we are going to use this interface for the rest of the metrics. This way, the plug-in tries to keep the Weka evaluation report tidy.

Each evaluation process has its own Evaluation object that will use the abstract methods defined in the `AbstractEvaluationMetric` class and the implemented interface to handle the plug-in metrics properly.

The plug-in metrics can compute the statistics using the `m_baseEvaluation` attribute of the abstract base class. This attribute is a reference to the Evaluation object. This way, the metrics have access to the protected fields of Evaluation.

3.1.3 Visualization plug-in

The graphic tool that gives the name to the package, the Entropy Triangle, can be used interactively with the classifications done in the Weka Explorer. Furthermore, we can load a file with results from another session.

A good place to call the plug-in to load these arff files is in the Weka GUI Chooser window that is launched at the program startup.

Explorer Classify panel

The Classify panel can be plugged to load visualizations with data from the performed classifications. The plug-ins are available through the right-click menu of the result history list. This sub menu, will show elements that fit the next two conditions:

- Reside in the package `weka.gui.visualize.plugins`
- Implement one of the interfaces of the package.

  Each interface is intended for a different kind of visualization and provides disparate information to build the visualization. We use the `ErrorVisualizePlugin` interface to get an updated copy of the test set instances.

The instances passed to the plug-in include an extra attribute with the predicted class value. We build a replica of the Weka Evaluation object of the process to compute the metrics for the Entropy Triangle.

Additionally, we extract the time and classifier with a reverse chain through the `ActionEvent` received on the plug-in call [37]. To this, we use the methods `getSource()`, `getParent()`, and `getInvorker()`, each where applicable, to reach a reference of the `ResultHistoryPanel`. Then, we call `getSelectedObject()` to get the reference of the classifier. With this information we identify unambiguously the elements in the visualization.

Extensions menu in the Weka GUI Chooser

The GUI Chooser window has a pluggable "Extensions" menu [31] that adds items with two requirements:

- The class of the menu component has to implement the `weka.gui.MainMenuExtension` interface.
- The `GenericPropertiesCreator.props` configuration file has to list the package of the class under the `weka.gui.MainMenuExtension` entry. This way, the automatic class discovery feature of Weka will find our class implementing the aforementioned interface. We have to create this file in the plug-in root directory.

The `weka.gui.MainMenuExtension` interface provides two ways to create the frame for our plug-in: the `fillFrame(Component frame)` method and registering an action listener. The first method receives the frame where plug-in will be placed, and the latter provides the plug-in with total control of what to do when an event is triggered. We use the action event because it allow us to share the same frame instance with the calls from the Explorer.

3.1.4 Package manager

The Weka package manager provides the user with a single interface to browse and manage the plug-ins [33]. However, the official package repository only stores the packages metadata. For this reason, we use a free hosting service for open source software. We intend to contact the Weka administrators to request them to link the package from their repository. In the meantime, the package can be installed from a compressed zip file.

For a plug-in to be installable through this system it must follow a certain structure.

Package description file

The `Description.props` file must be included in the package root directory. Otherwise, the package manager will not install the plug-in. If the package is available through the official repository, this file will be used to update package metadata and advise for updates. Therefore, is important to update the "Version" field on each release of the plug-in.

A complete explanation of the `Description.props` fields can be found in the Weka manual [3] as well as in the Weka wiki [32] including a template in [30].

Here, we will just mention that the required fields are "PackageName","Version", "Title", "Author", "Maintainer", "License", "Description", "Depends", and "PackageURL". The rest of them are optional, but the "Category" field is recommended. Besides, there is a field for an URL for further information where we can link to the project page.

The "Category" field is a free string, we mark "Classifiers evaluation" and "Visualization". This way, we remark that the package contains a visualization tool. The plug-in metrics are embedded in the evaluation part.

Package structure

The plug-in packages for Weka have to be packed in a zip file and follow a particular directory structure.

A build automation tool is recommended to compile the Java source code and bundle the plug-in. In the Weka wiki there is a template file [29] available for building the package with Apache Ant, `build_package.xml`, that we have modified. The file, as well as the plug-in source code, is included in the plug-in zip package. The main modifications we made on the template are:

- Set project name to "EntropyTriangle", to automatically generate the package name.
- Change Java version from 1.6 to 1.7, the same version used in Weka 3.7.12
- Add an *include* directive to copy all *props* files present in the work folder to the root folder of the generated file. The template only adds the `Description.props` file to the package, but we also need to add the `PluginManager.props` (Section 3.1.2) and the `GenericPropertiesCreator.props` (Section 3.1.3) files.
- Add an *include* directive for *txt* files in the root directory. This way files like `README.txt` or `LICENSE.txt` will be added automatically.

Java source files are usually organized in a path that follows the structure of the Java packages they belong [20]. Moreover, a dedicated directory, commonly called `source` or `src`, can be on top of this folder hierarchy to separate source files from compiled code, that is then targeted to a `build` folder. The `build_package.xml` template introduces another level of abstraction for the source folder hierarchy, the functionality files goes in the `main` folder just after `src`, and optional test files can be included in the `test` branch. The directory skeleton of the Entropy Triangle package is summarized in Table 3.2.

```
<working directory>
  +-Description.props
  +-PluginManager.props
  +-build_package.xml
  +-src
  |   +-main
  |   |   +-java
  |   |       +-weka
  |   |           +-etplugin
  |   |           |   +-EntropyTrianglePanel.java
  |   |           |   +-TernaryPlot.java
  |   |           |   +-metrics
  |   |           |       +-Ema.java
  |   |           |
  |   |           +-gui
  |   |               +-visualize
  |   |                   +-plugins
  |   |                       +-EntropyTrianglePlugin.java
  |   +-test
  |       +-java
  |            +-weka
  |               +-etplugin
  |                   +-TestTriangle.java
  +-lib
  +-doc
```

Table 3.2: Entropy Triangle package directory structure (Adapted from [32])

3.2 System Design

The design stems from the Weka interfacing options and their requirements, revised in Section 3.1. From this point, we have analyzed different options for the components of the plug-in, and tried to choose those that fitted best with the global design and individual tasks. There are some configurable values at several places of the program, like the plot grid, that has been hard-coded for simplicity. Future versions may consider introducing a configuration file that let the user customize appearance and options.

Overall, we have pursued an object-oriented hierarchical and modular design. For that, we used inheritance for a clean and highly abstracted implementation, and a hierarchical organization of components for a better isolation of scopes. Other essential considerations include building a program as robust and lightweight as possible. We also took advantage of the potential offered by Weka and Java for several tasks.

The design of the package has been planned in advance, although the implementation of the software classes and functions has been done incrementally. We implemented the evaluation metrics in the first place as they are needed for the Entropy Triangle visualization. Then, we continued with a basic ternary plot making it callable from Weka. Then, we gradually added functionality until reaching the proposed objectives. We think that this approach has driven us to a better final design because the project was implemented from the basic functions, but having a concept of the entire goal from the beginning. This way we could evaluate the impact of each step, and consider redesigns when appropriate.

3.2.1 Package structure

The Java classes of the plug-in are grouped in two Java packages: `weka.etplugin` for the Entropy Triangle elements and panels, and `weka.etplugin.metrics` for the evaluation metrics. The only exception is the `EntropyTrianglePlugin` class, that handles the calls from Weka to the Entropy Triangle. This class, as explained in Section 3.1.3, has to belong to `weka.gui.visualize.plugins` package. Figure 3.1 shows a class diagram that outlines the `weka.etplugin` package.

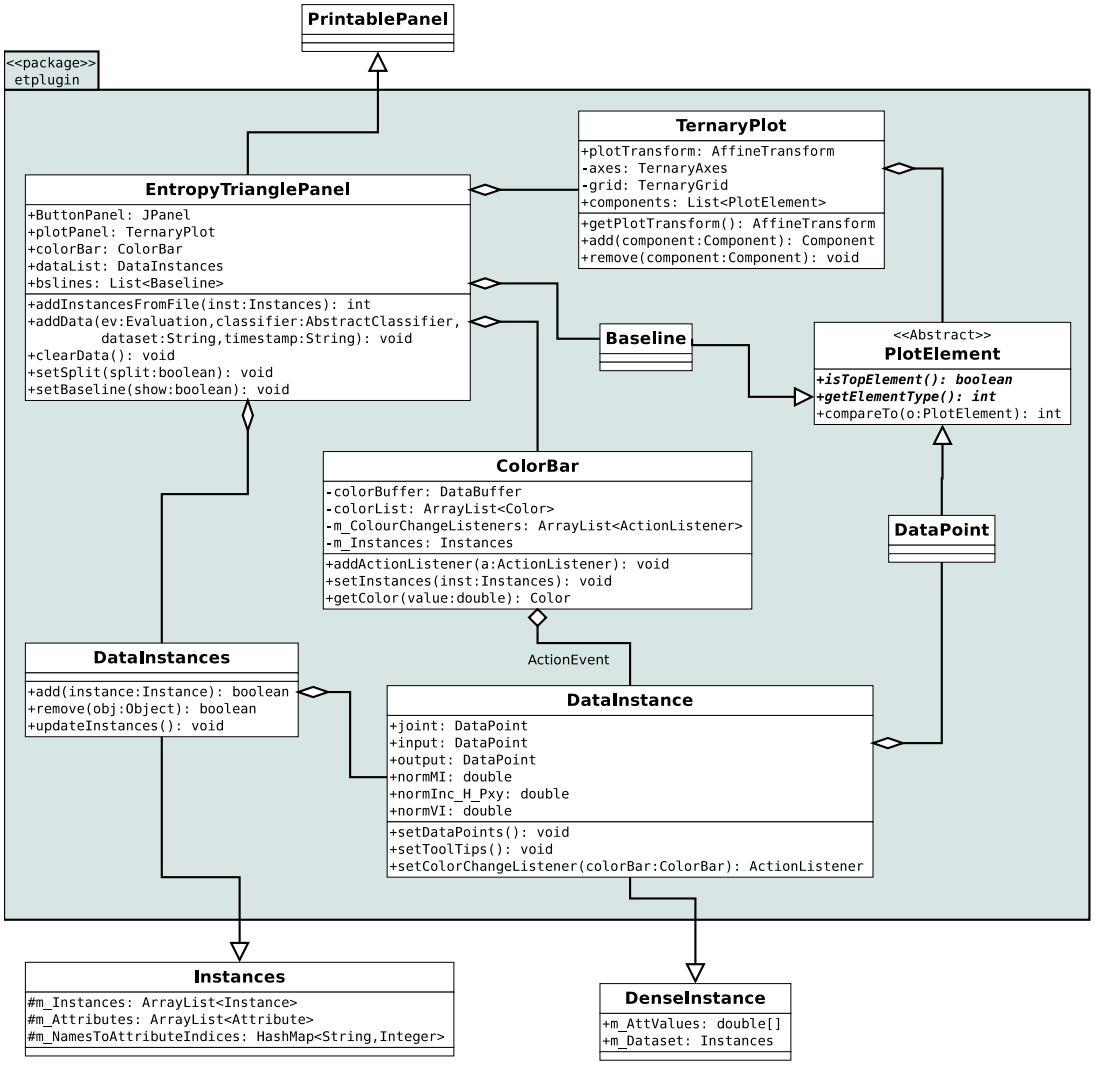


Figure 3.1: `etplugin` package class diagram

3.2.2 Entropy Triangle

The main class of the plug-in is the `EntropyTrianglePanel`. This panel acts as coordinator of the program flow and top level container of buttons, ternary plot and color bar panels. Fig. 3.2 shows the Entropy Triangle with data from five experiments. Some key tasks that the Entropy Triangle panel does as object interaction coordinator are:

- Initializing and implementing the buttons logic, including file loading and saving.
- Holding `DataInstances` and `Baseline` lists.
- Transforming raw content from the Explorer to the `DataInstance` format.
- Creating `Baseline` objects for the $\Delta H'_{P_X}$ values of the data added to the plot, or adding labels to the corresponding `Baseline` when the value is very close to an existing one.
- Mediating between the `DataInstance` objects and `TernaryPlot` to add or remove points in the plot.
- Registering the color change listener of the `DataInstance` objects in the `ColorBar`.

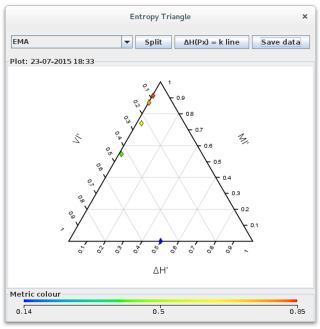


Figure 3.2: Entropy Triangle window

3.2.3 Ternary Plot

The plot is divided in different Java classes where each one represents a graph element. The `TernaryPlot` class is the Java panel that contains the plot elements as Java children. The elements handle their own functionality, like the response to mouse events for showing the tooltips or, in the case of the `Baseline` opening the dialog for changing its color. In the plot, the represented objects are independent among them and only handled by the ternary plot panel or objects that contain them as attributes, in a higher level of abstraction (see Sections 3.2.2 and 3.2.5).

The `TernaryPlot` panel use the `AffineTransform` class of the Java package `java.awt.geom` to provide a graph coordinate system that simplifies the design. The other main function of the panel is to display the elements hierarchically in order of importance, necessary for a better visualization and interaction of the tooltip windows of the elements. This is easily done with the `Comparable` interface and the `Collections` sorting method, both from the Java standard libraries.

Affine transform

Java provides an abstraction for the coordinate system in which *Container* objects have their independent logical origin [13]. The top-left corner of the container is the origin and the coordinates are integers mapped to screen pixels. This means that the size of the objects depends of the screen resolution.

We want to display the entire diagram in the window, at the maximum size possible. In addition, we want to transform the Java coordinates to a more natural system for the plot, shown in Fig. 3.3. The Java class `AffineTransform` (from the package `java.awt.geom`) provides methods to scale, rotate, invert, and apply general affine transformations [13, 24].

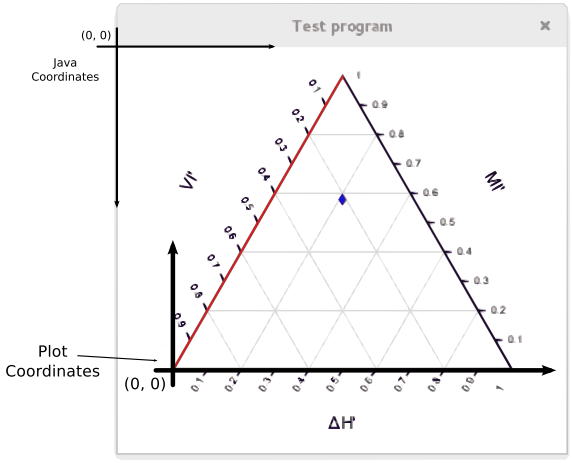


Figure 3.3: `TernaryPlot` testing program

We considered the alternative of mapping the coordinates using custom methods, but we realized that the `AffineTransform` class outperforms any implementation we could design and facilitates possible future modifications. Besides, we get a cleaner design easier to understand by other programmers.

The final implementation scales the plot coordinates to a normalized range $(0, 1)$ with extra margins on both sides. Then, it adds an affine transformation that ends the conversion. On the one hand, it inverts the y axis and adds an offset to the vertical origin. On the other hand, it centers the plot horizontally fixating the origin at a distance of 0.5 to the left of the center. Once this is done, we can forget about Java constraints and design the diagram naturally.

However, a few issues concerning the appearance of tooltip windows on mouse hover should be taken into account. Tooltips are shown based on the returned value of the `contains(x, y)` method of the graph components. This method is called automatically by Java receiving the coordinates in Java user space format. We map this coordinates overriding the method and obtaining the plot coordinates. For that, we store the affine transform in the `TernaryPlot` and provide a getter method. Plot elements can access the `TernaryPlot` via the `getParent()` method and then, get the current affine transformation to translate the coordinates. Every time the panel changes its dimensions, the affine transformation is computed and stored.

Plot elements

Every element in the graph is a different object. The `TernaryPlot` panel mission is to draw these elements in an appropriate stacking order at their coordinates. For instance, points representing evaluation data, or the lines that represent a $\Delta H'_{P_X}$ value, have to be drawn on top of the grid and axes. Though this may not seem an issue, since the grid and axes are added first, the stacking problem arises with the dinamic use of the plug-in, for instance, when we add a data point after having drawn the $\Delta H'_{P_X}$ lines. To avoid this unwanted situations, we base the different types of elements in a joint abstract class, `PlotElement`, that implements the `Comparable` interface of the Java standard libraries.

`PlotElement` implements the logic to sort the elements based on their type, defines a constant for each element type assigning different integers to them. The `TernaryPlot` sorts its list of components with the static method `sort(list)`, of the `Collections` class, before painting them.

In order to handle cleanly dense populated graphs, we added an extra feature to the `PlotElement` class. The `PlotElement` can have two layers of components. Thereby, components of the top layer, *TOP_ELEMENT* components, will be drawn over regular elements.

Classes that extend `PlotElement` only have to override two methods to return their type: `isTopElement()` and `ElementType()`. The former returns a boolean indicating their priority and the latter, an integer with the value of the respective type of element constant. Table 3.3 shows the values assigned to the constants.

Table 3.3: *`PlotElement` constants*

Constant	Value
TOP_ELEMENT	4
POINT_ELEMENT	-1
LINE_ELEMENT	-2
AXES_ELEMENT	-3
GRID_ELEMENT	-4

Axes and grid

Given the specific purpose of the visualization, the `TernaryAxes` class is designed as simple as possible, the same class is used for the three axes with the ticks and axes labels configuration hard-coded. In the `TernaryGrid` class the grid distance is fixed as well.

Both classes are just wrappers to draw the items on the panel and could have been joined, but since the axes class is a bit large, the grid was considered a natural object to separate. The code gets cleaner because the grid uses different stroke than the axes.

For rotating the text in the tick and axes labels we implemented a static method that takes as arguments the `Graphics` object, the text to draw, the rotation angle, and the bottom-left coordinates where the text should be drawn in the screen. The method clones the `Graphics`, the brush, rotates it, writes the text with the rotated brush, and finally discards the brush. This way we isolate the rotation from side effects on object elements. An alternative that we considered, was to rotate the text, and draw it rotated with the regular brush, but we found that this approach was not acceptable on Mac computers for aesthetic reasons.

DataPoint

The `DataPoint` is a simple class that basically draws a diamond at the coordinates passed to the constructor. This class extends `PlotElement`, and overrides the `contains()` methods of `JComponent` to handle the mapping of mouse coordinates, in the Java user space system (Section 3.2.3), to graph values.

Changing of the foreground color, and the text in the tooltip, is handled by the `DataInstance` class. The design goal of the data points was that they should only be generic points in the graph: the semantic of the point is better handled at another abstraction layer.

Baseline

In contrast with the data point class, the `Baseline` class handles itself its entire information and functionality. The reason for this different approach is that baselines do not have a natural ancestor at their upper layer.

This class has a string list whose elements identify the performed evaluations that the line represents. When the data point of a classification is removed from the graph, the `EntropyTrianglePanel` calls the `removeLabel()` method of the corresponding `Baseline`. The method returns the number of strings remaining in the list after the removal. This is a key element for the removal of the baseline of the plot when the line does not represent any evaluation on the graph.

In a similar fashion that the colored labels of the Weka `ClassPanel`, *clicking* the line opens a `JColorChooser` dialog that let us change the line color. The tooltip is updated on list changes by private methods.

3.2.4 Color bar

The `ClassPanel` of Weka has the functionality we need, draws a gradient to differentiate the element values and place colored labels for the text characteristics. The Weka instances format fits perfectly well to interact with the class panel. In addition, it provides to the visualization the Weka look-and-feel. For these reasons, we used at first the `ClassPanel` as color bar. In order to customize this panel, we created the `ColorBar` from its sources.

We wanted to add the heatmap effect of a rainbow style gradient (see Fig. 3.4). At first, we considered the extension of the class to override the colorizing of the panel, but the necessary methods were protected. With the intention to keep the plug-in classes in the same Java package, we decided to customize the class from a copy. Another issue was how to extract the color value from the class panel, the new class solves it storing a buffer array with the color values and a method to map metric values to buffer positions. The `ColorBar` extracts a raster creating a `PaintContext` (in the `java.awt` package) object of the bar coordinates, and from the raster, the color data buffer of the gradient. The `LinearGradientPaint` class of the `java.awt` package is used for painting the gradient.

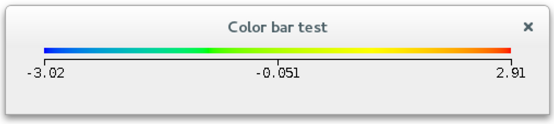


Figure 3.4: Color bar testing program

3.2.5 Data structure

A backbone of objects interaction is the main data structure. Getting an optimal design is tricky, and often only achievable after some attempts. Fortunately, the software design can be restructured.

The confusion matrix could be computed from the predicted instances that Weka provides to the plug-in, but this scheme means computing again all the metrics. From that, one can conclude that storing the `Evaluation` object (see, Section 3.1.3) is a better idea. However, this entails calling different methods to retrieve each metric value, somehow blurring the code.

Another option is to retrieve the metric values before adding the data to the structure. This approach can be implemented efficiently, in terms of memory, with coordinated lists: a list that links the metrics names, or labels, to their positions in the list can be shared storing it in the higher level object. Then, the wrapper objects of the evaluation data and graph points only need to store the values list in the same order that the labels. This data structure has been used during the program development, and was intended to be the final scheme.

Opportunely, when seeking how to write the data to a file with the tools that Weka offers, we realized that the instances format was central for using these tools. In fact, what we needed to do, was already done in the Weka Experimenter and was documented in the code. The instances format provided us with the possibility to outsource all the data handling with the Weka API. This format implements hash tables, that apart from being memory-efficient are also computationally efficient.

Still, we needed to create a wrapper class to handle the plot data points, to change their color or tooltip, to add or remove them in a coordinated manner and to store the triangle entropies. So, we extended the `Instance`

class of Weka with the `DataInstance` class, inheriting the efficient structures and incorporating the attributes we needed for the Entropy Triangle. We had to override the `copy()` method called by some routines for casting instances to the derived class. For the same reason, we had to extend the `Instances` class to avoid type conflicts. The new derived set of classes interacts correctly with the Weka API for file writing and reading.

In the background, setting the class attribute for the metric chosen to colorize the graph points simplifies the interaction. `DataInstance` objects create action listeners that the `EntropyTrianglePanel` binds to `ColorBar` events. Then, the `ColorBar` fires events for any modification that affect colors, and finally, the listeners call the `getColor()` method of the bar with the value of their respective class attribute.

3.2.6 Evaluation metrics

The scheme of the pluggable evaluation metrics is constrained by the Weka interfacing design. The options available are the way to compute the metric value and the output string to include the metric in evaluation reports.

All the package metrics can be computed from the confusion matrix. The pluggable metrics have a reference to the base `Evaluation` object from where we get a copy of the confusion matrix with the method `confusionMatrix()`. The entropies are computed via the static methods of the Weka class `ContingencyTables`, and the perplexities with the `Math` class of the Java standard libraries. To illustrate, Fig. 3.5 shows the class diagram for the `InformationTransferFactor`.

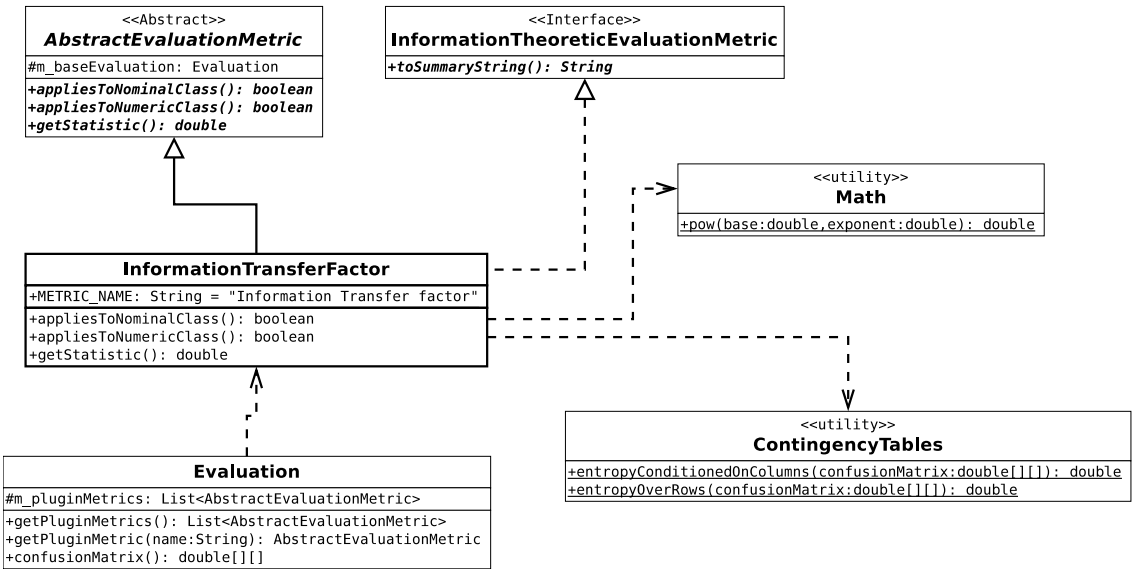


Figure 3.5: Class diagram of `InformationTransferFactor`

For the text output format, we use the same style as the Weka metrics, i.e, title case, leaving in lowercase common words like statistic, factor, or error. Although Weka uses *bits* for the entropic measures, we do not consider relevant to include a metric unit since the intended user should know the information it represents. The EMA and NIT as well could be displayed as a percentage, but we consider that the authors intention were to use them as originally designed: normalized values in the 0-1 range.

Metrics do not perform any other function themselves. The Evaluation object associated with each classifier test has a method that returns the plug-in metric objects of that process by the metric name. We use that method to obtain a reference of the metrics. And then, we call their `getStatistic()` method to get their value for the Entropy Triangle.

3.2.7 Loading the visualization plug-in from Weka

In Section 3.1.3 we already argued that the Weka GUI chooser window is a good place for loading data from files into the visualization tool. When called from the Explorer, we want the data to be added to the existing graph, or create a new one if necessary. The available options for handling the Weka calls to the plug-in are either to combine calls from both access points to the same plug-in instance or creating separate windows for each source.

After considering both scenarios, we decided that joining all the calls in the same graph avoided possible user confusions due to multiple windows. Therefore, we use the `EntropyTrianglePlugin` class (see Fig. 3.6) for both entry points, and declare the `JFrame` and `EntropyTrianglePanel` instances static. The panel data is cleared when the user chooses to close the window. Then, if the user opens the plug-in again, the graph will be empty. We think this scheme provides a better user experience.

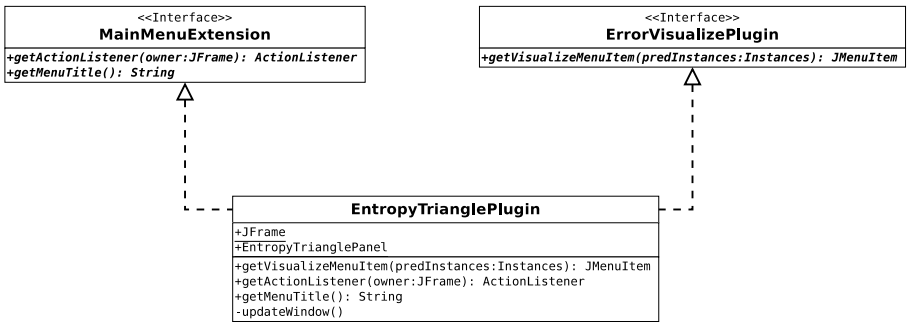


Figure 3.6: Class diagram of `EntropyTrianglePlugin`

For simplicity, the current implementation toggles the upper-right button to save or load data depending of the context. If the graph is empty, called from the GUI Chooser, the button will be active as a load command and adding saved data is enabled. When the plot contains data, the only possible option should be to save it and therefore the button toggles accordingly. This behavior may be redesigned in future revisions with dedicated buttons for each command.

3.2.8 Weka as library for the plug-in

Producing a lightweight plug-in and reusing as much tested code as possible have been leading aspects for the implementation. Functionalities extracted from the Weka API [25] and the Java Class Library (JCL), standard Java packages included with the Java Virtual Machine (JVM) [20], provide efficient code without having to include it in the package. With this in mind, we performed a thorough search upon the Weka API and source code to identify how Weka performs calculations, graphic charts, and data handling. We identified useful classes and methods for several complex tasks that we needed to implement for the package. In this section, we summarize the Weka classes chosen to fulfill the plugin functionality requirements. The classes used by requirement to interface Weka are described in Section 3.1.

weka.core.ContingencyTables
: The entropies are computed with static methods provided by this utility class for matrices.

weka.core.Utils
: This class provides utilities, static methods, that we use to convert double numbers to strings, compute binary logarithms, and merge string arrays. We also use this class to check if an attribute value is missing keeping the level of abstraction that Weka provides, avoiding possible future incompatibilities due to changes in their coded values.

weka.gui.visualize.PrintablePanel

The `PrintablePanel` lets us save the complete window of the graph with a simple keyboard and mouse command. We have chosen this option over other alternatives because it maintains the same commands as Weka graphs and its easy interfacing. We only need to extend this class with the graph panel and all the functionality is inherited. Moreover, we avoid including external libraries.

weka.core.Instances

A key component of the Entropy Triangle is the internal data structure of the evaluation data. After trying some custom designs, we finally realized that the best suitable option was to benefit from the Weka's Instances format. The implementation is based on how Weka Experimenter uses `Instances` for saving evaluation data. In spite of not being compatible yet, this scheme eases the possibility of loading files produced by the Experimenter on the Entropy Triangle, and vice versa. In addition of being computationally efficient, the instances format provides tools for reading and writing files in several formats with specialized dialogs.

weka.gui.visualize.ClassPanel

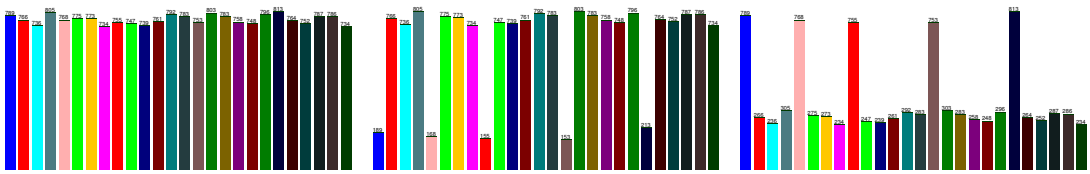
The Entropy Triangle can include a color bar to determine the foreground color of the graph items based on their value for a given metric. Weka implements this functionality for the graphs in the `ClassPanel`. The Instances format makes our data directly compatible with the `ClassPanel` class. As we explain in Section 3.2.4, this class uses a gradient that directly passes from blue to red in the RGB color map. In order to show a gradient-like color spectrum it was necessary to modify non public methods of this class, and therefore, this was not achievable trough the API. Then, observing the terms of the Weka license [11], we modified the code to implement the gradient-like spectrum with the Weka *look and feel*.

Chapter 4

Use cases

4.1 Interactive Use

In this section we show how the Entropy Triangle works. To evidence the effect of unbalanced datasets on the metric values we created two subsets with unbalanced class distribution from the Letter dataset, 26 classes corresponding to the letters of the alphabet [7]. We removed 600 instances of each vowel (Fig. 4.1(b)) in one of them, and 500 instances of each consonant (Fig. 4.1(c)) in the other.



(a) Complete dataset (b) Removed vowels dataset (c) Removed consonants dataset

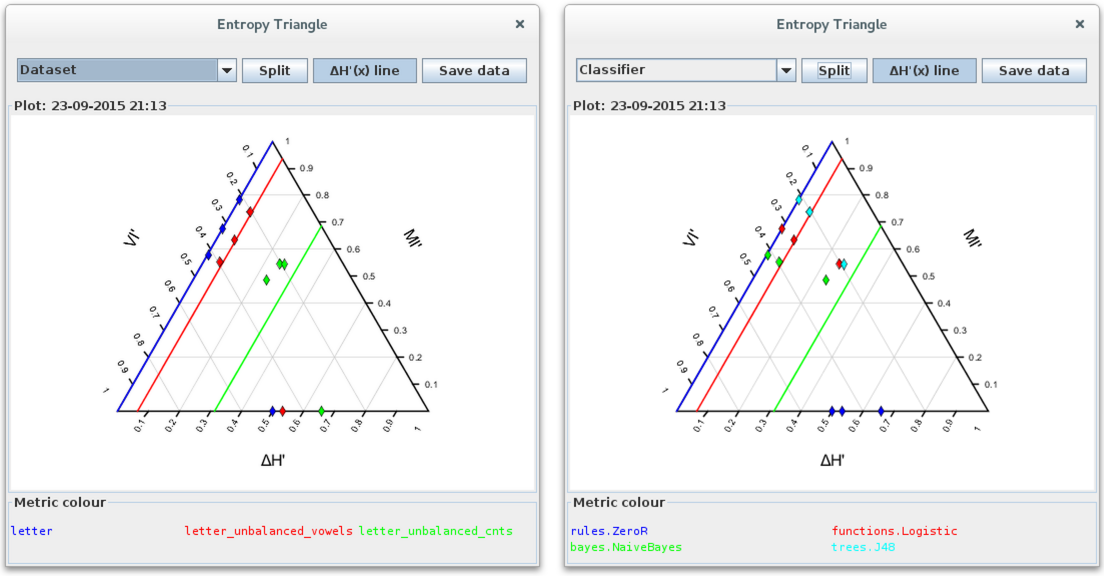
Figure 4.1: Letter dataset class distribution

For this example we use the following classifiers: zero rules, logistic regression [14], naive Bayes [12], and C4.5 [19] (J48 in Weka). The classifiers are trained with the 70% of instances and evaluated with the remaining 30%, all with their default parameters. Figure 4.2(a) shows the Entropy Triangle with the results colorized by dataset and including the ΔH_{P_X} lines.

4.1.1 Identifying data in the Entropy Triangle

The data in the Entropy Triangle can be identified interactively by means of the tooltips on mouse hover. Besides, we can have an overview selecting the dataset or classifier in the combo box to colorize the data, like in Fig. 4.2. Choosing the classifier name to colorize the data (Fig. 4.2(b)) we can appreciate that C4.5 seems to outperform the other classifiers for the balanced dataset and the one with removed vowel instances.

In the dataset with removed consonants the C4.5 and logistic classifiers seems to have equal mutual information (MI). They difference in terms of variation of information (VI) or increment of entropy from the uniform distribution (ΔH) does not seem to be significant as well. The zero rule classifier shows a bad performance for the three datasets. This happens as expected because none of the sets has a single majority class.



(a) Data colorized by dataset

(b) Data colorized by classifier

Figure 4.2: Entropy Triangle window with different classifiers for the unbalanced letter datasets

4.1.2 Using the evaluation metrics and the split view

To characterize the performance of the classifiers through evaluation metrics we remove the zero rule results from the Entropy Triangle. This way, the colorbar range is shorten and differences in closer values are better appreciated. Fig. 4.3 shows the data colorized by the accuracy and Fig. 4.4 by the entropy modulated accuracy (EMA).

We can see that the EMA values for the logistic and C4.5 classifiers in the removed consonants dataset are higher than in the rest of cases. This value is explained because EMA is the inverse of the remaining perplexity, and for this dataset the classifiers have achieved a higher reduction of perplexity than for the other more balanced datasets. The split mode of the Entropy Triangle shows also this improvement with the P_Y markers of these classifiers on the left of the P_X points (Fig. 4.5).

4.1.3 Unbalanced evaluation of balanced trained classifiers

The maximum values of the mutual information are limited by the class distribution of the test sets. We identify this limit in the Entropy Triangle with the ΔH_{P_X} lines. To analyze this constraint we performed evaluations with the unbalanced sets of the classifiers trained with the balanced set. We changed the baseline from the zero rule classifier to the one rule classifier [10].

Fig. 4.6 shows the results, that are quite similar to the case with unbalanced training. The only classifier that improves is the C4.5 for the removed constants dataset, showing that with a balanced training can predict better. Anyway, the values are limited by the class distribution of the test set.

These results give an insight of how to interpret the values in the Entropy Triangle. Performance of classifiers is determined by their training data and learning scheme, but the testing conditions may influence their scores. Therefore, there are necessary tools for a reliable assessment, independently of the data available for evaluation. The Entropy Triangle and the metrics of the package provides an information-theoretic framework for the assessment of classifiers that take into account the context in which they are evaluated.

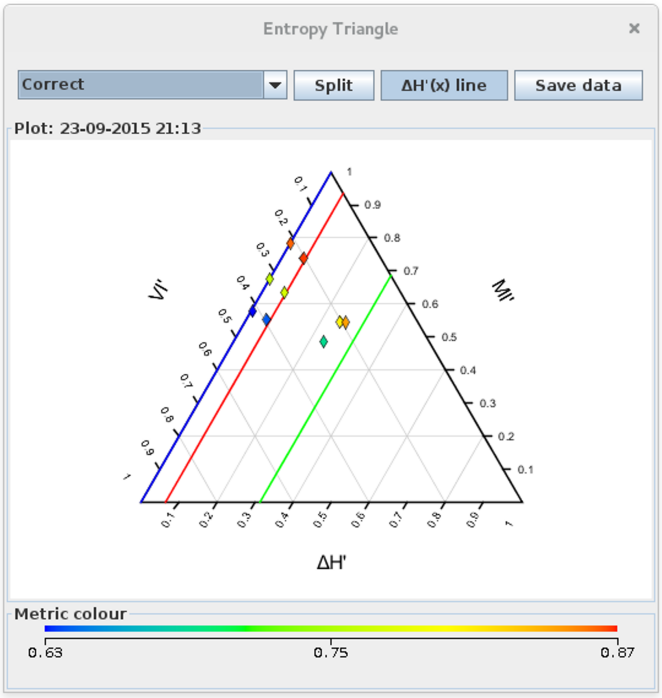


Figure 4.3: Entropy Triangle with values colorized by accuracy

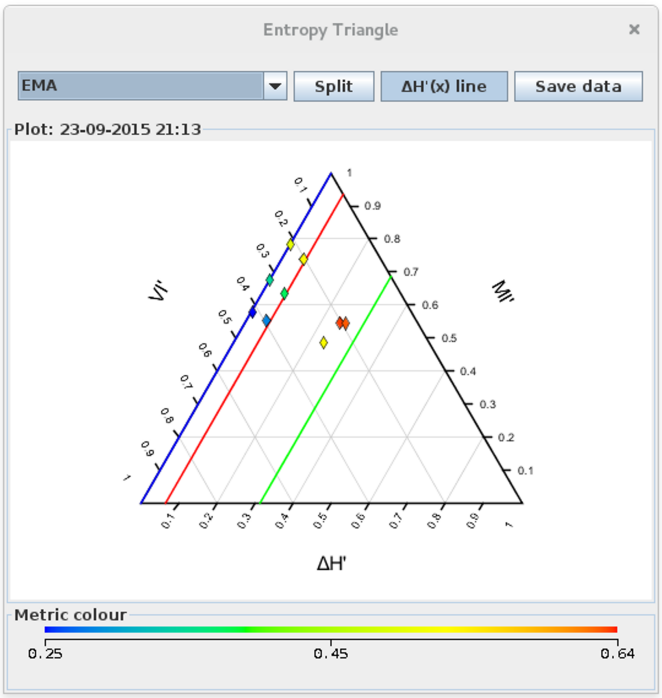


Figure 4.4: Entropy Triangle with values colorized by EMA

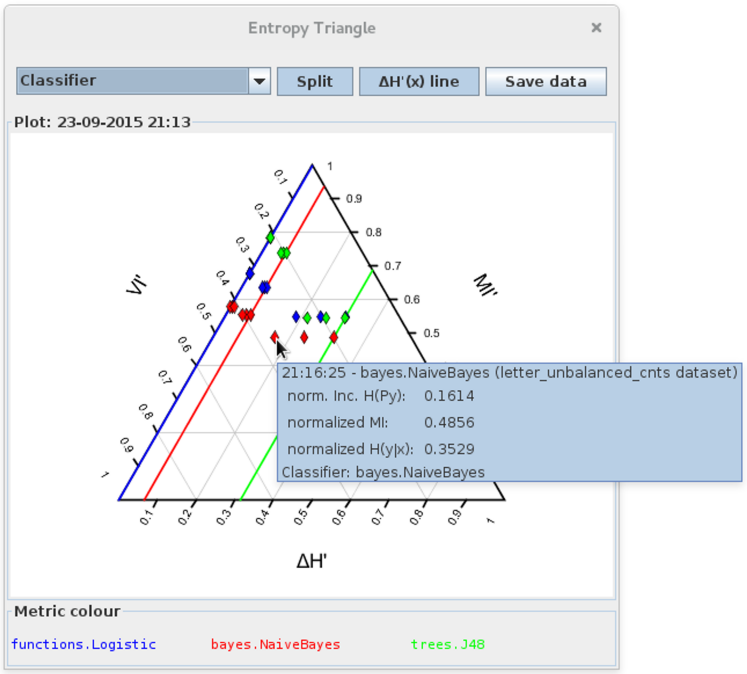
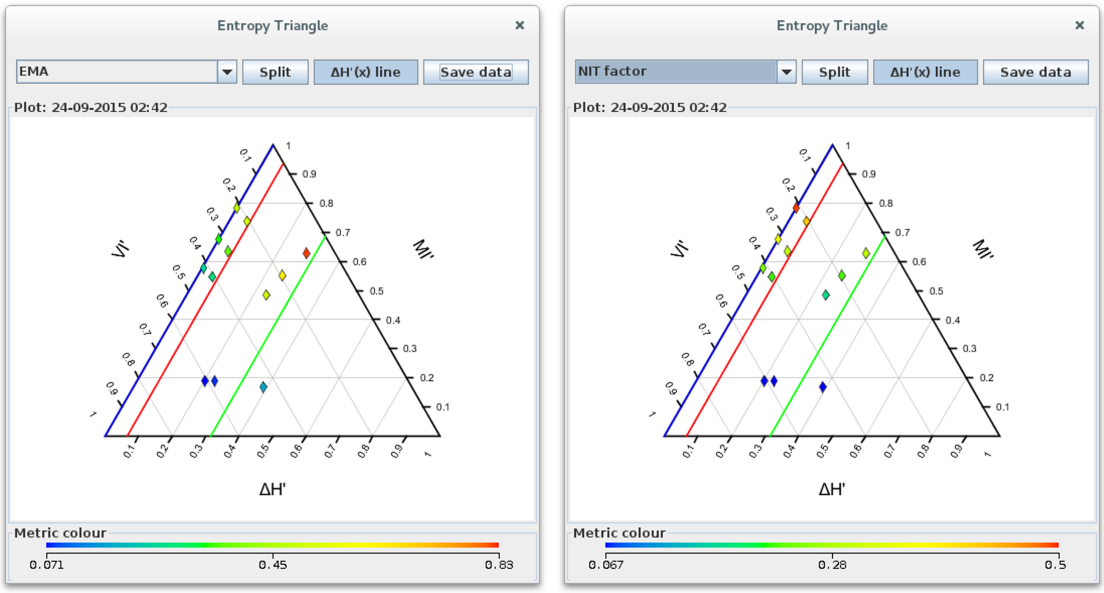


Figure 4.5: Entropy Triangle in split view



(a) Data colorized by EMA

(b) Data colorized by NIT factor

Figure 4.6: Entropy Triangle with unbalanced evaluation of balanced trained classifiers

4.2 Programmatic use

A good way to configure a Weka experiment is through Java code, with only few lines of code we can setup a complete experiment. This approach let us play with the data modifying only the needed parts, and conserve the experiments configuration to reproduce them.

Both, the Entropy Triangle visualization and the new metrics can be used through code. The next section shows a simple example of how to add data to the Entropy Triangle, and the last section how to print an evaluation report including the package metrics.

Additional requirement

This method requires the Java Development Toolkit (JDK) to compile the source files.

4.2.1 The Entropy Triangle from code

In this example we are going to train and evaluate four classifiers with the segment dataset that is included with Weka. This dataset comes already splitted in two files for the train and test sets. Finally, we add the evaluation data to the Entropy Triangle to use it interactively.

All the code of this example goes in the same file, MyExperiment.java. We divided it in several boxes for illustration.

For a more detailed explanation on using Weka with code, see the Weka wiki pages for programmatic-use[1] and how to use Weka in your Java code[2]. Also, the Weka API of the developer version (3.7).

MyExperiment.java

We create an empty text file and name it as our Java class, i.e. `MyExperiment`, appending `.java`. In this file, we are going to define a Java class with only the `main` method. Writing all the instructions inside the main method will make them run sequentially.

First of all, we have to create the Entropy Triangle panel, and a window, a `JFrame`, to place it.

```
1  import javax.swing.JFrame;
2
3  import weka.core.Instances;
4  import weka.core.converters.ConverterUtils.DataSource;
5  import weka.classifiers.evaluation.Evaluation;
6  import weka.etplugin.EntropyTrianglePanel;
7
8
9  public class MyExperiment {
10
11     public static void main(String[] args) {
12
13         JFrame frame = new JFrame();
14         EntropyTrianglePanel et = new EntropyTrianglePanel();
15
16         frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
17         frame.add(et);
```

[1] https://weka.wikispaces.com/Programmatic+Use
[2] https://weka.wikispaces.com/Use+Weka+in+your+Java+code

```
18          frame.setVisible(true);
19          frame.pack();
20
21    **[CODE from Section: "Load the dataset, train the classifier and evaluate..."]**
22
23      }
24  }
```

Load the dataset, train the classifier and evaluate the model

Once the Java environment is configured, we can start with the Weka instructions.

Some Weka methods throw Java exceptions if something does not go as expected. Therefore, we will surround our code in a try block with a basic catch to print the error trace.

First, we have to load the train and test sets of instances. We set the index of the class attribute to the last one, we can skip this if the `arff` files already has defined the class attribute.

Once we have the data loaded we are going to test it with a `ZeroR` classifier to have a baseline. The procedure is as follows:

1. Create an `Evaluation` object initialized with the prior probabilities of the train set.
2. Create a `ZeroR` classifier object. We use the fully qualified name for classifiers to avoid handling imports when changing of classifiers.
3. Train the classifier with the train set.
4. Evaluate the classifier for the test set.

```
1  try {
2    DataSource source = new DataSource("./datasets/segment-challenge.arff");
3    Instances train = source.getDataSet();
4    Instances test = DataSource.read("./datasets/segment-test.arff");
5    train.setClassIndex(train.numAttributes() - 1);
6    test.setClassIndex(test.numAttributes() - 1);
7
8    Evaluation eval = new Evaluation(train);
9    weka.classifiers.rules.ZeroR zr = new weka.classifiers.rules.ZeroR();
10   zr.buildClassifier(train);
11   eval.evaluateModel(zr, test);
12
13   **[CODE from Section: "Add data to the plot"]**
14
15 } catch (Exception e) {
16   System.out.println("Error on main");
17   e.printStackTrace();
18 }
```

Now, we have the results stored in the `Evaluation` object and ready to be added to the Entropy Triangle.

Add data to the plot

The manager of the Entropy Triangle data is the `EntropyTrianglePanel` object we defined previously. To add evaluation data to the visualization we only have to call the addData() method of this class.

The `Evaluation` object and the classifier are passed as the first two arguments. The third argument is a string used to identify the dataset, we use the dataset relation name for consistency. The last argument is another string used for time-stamp information; the experiment execution time is used if the argument is `null`.

```
et.addData(eval, zr, test.relationName(), null);

**[CODE from Section: "Adding more data"]**
```

Adding more data

The main advantage of the Entropy Triangle is that lets you compare easily different dataset-classifier setups. We can proceed similarly to add the evaluation of other classifiers, or try different datasets, either loading different `arff` files or applying Weka preprocessing filters to the `Instances` objects.

In this experiment, we are going to compare different classifiers with the same dataset. For that, we repeat the instructions we used before for the ZeroR classifier. The train and test `Instances` objects are used only for reading the instances, so can be safely reused. The `Evaluation` object, eval, is overwritten with new object that only has the train set prior probabilities.

We are going to test the OneR, NaiveBayes, and J48 (C4.5) Weka classifiers with the default parameters.

```
eval = new Evaluation(train);
weka.classifiers.rules.OneR oner = new weka.classifiers.rules.OneR();
oner.buildClassifier(train);
eval.evaluateModel(oner, test);
et.addData(eval, oner, test.relationName(), null);

eval = new Evaluation(train);
weka.classifiers.bayes.NaiveBayes nb = new weka.classifiers.bayes.NaiveBayes();
nb.buildClassifier(train);
eval.evaluateModel(nb, test);
et.addData(eval, nb, test.relationName(), null);

weka.classifiers.trees.J48 j48 = new weka.classifiers.trees.J48();
j48.buildClassifier(train);
eval = new Evaluation(train);
eval.evaluateModel(j48, test);
et.addData(eval, j48, test.relationName(), null);
```

Running the experiment

To compile and run the experiment we have to include in the classpath the `weka.jar` and `EntropyTriangle.jar` files. We can append the -classpath option to the javac and java commands or append the files to the classpath variable of the terminal session.

Setting the classpath

- Linux / Mac

```
$ export CLASSPATH=${CLASSPATH}:<path-to>/weka.jar
$ export CLASSPATH=${CLASSPATH}:${HOME}/wekafiles/packages/
    ↪ EntropyTriangle/EntropyTriangle.jar
```

- Windows

```
> set CLASSPATH=%CLASSPATH%;.;%PROGRAMFILES%/Weka-3-7/weka.jar
> set CLASSPATH=%CLASSPATH%;%USERPROFILE%/wekafiles/
    ↪ packages/EntropyTriangle/EntropyTriangle.jar
```

Compile and run

```
# Compile the experiment
$ javac MyExperiment.java

# Run!!
$ java MyExperiment
```

Running the program produced with the experiment opens the Entropy Triangle window of Fig. 4.7. The figure shows the tooltip for the one rule classifier. The user can explore the results interactively with the mouse, and change the displayed information with the window buttons.

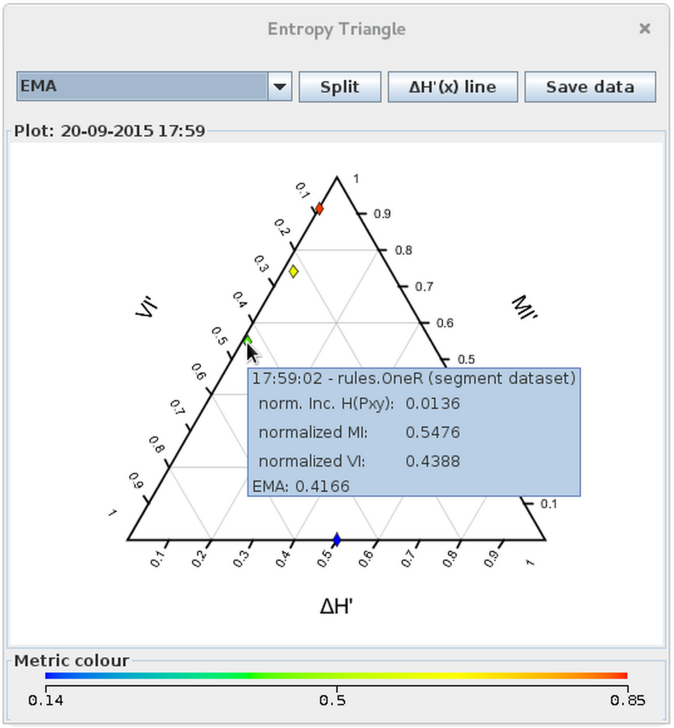


Figure 4.7: Entropy Triangle produced by `MyExperiment.java`

4.2.2 Printing the plugin metrics

The plugin metrics are integrated in the Weka `Evaluation` class. This makes them available on all the interfaces, including the output from code. To print a classification report, like in the Weka explorer, we have to call the method `toSummaryString()` of the evaluation object. For including the information-theoretic statistics in the report, we have to call the method with `true` as argument: `toSummaryString(true)`.

Chapter 5

Conclusions

There are projects that can extend for long in time. Software developments are often affected by that: there are always things that can be either improved or added. Therefore, it is wise to split them in stages. The Entropy Triangle Package for Weka is one of that kind of project. In this thesis we have documented the first phase. This chapter describes the conclusions we have drawn from the project. Besides evaluating the work of this months, we outline what we consider the natural next steps.

5.1 Software development

We proposed to implement the complete set of information-theoretic tools described in [22, 23] as a plug-in for the Weka machine learning suite. The first release of the program that we present here meets this main requirements successfully. We have produced a working program that can be easily installed through the Weka standard procedure for packages. Moreover, it is accessible while unobtrusive.

An important aspect, mathematically speaking, is that we have outsourced the complicated calculus to efficient and tested classes. The metric computations are made through Weka methods. Except the calculation of perplexities from the entropy, but this is done with the `Math` class of the Java Class Library. As a result, we have a more robust program.

The architecture designed for the package has a high modularity and a hierarchical structure. The Java classes represent concrete elements of the program, and their concept try to follow the object-oriented scheme. For instance, we have abstracted common functionality to base objects, e.g. the `PlotElement` abstract class for ordering objects in the graph. Similarly, we interfaced and extended features available in Weka and the Java standard libraries to produce better software with less complexity and code, like extending the Weka `PrintablePanel` for printing the window to an image.

From the user perspective, the entropy triangle is highly interactive. The graph is kept tidy but contains complete information. There are two key components in this result: the button bar to manage the graph content, and the use of tooltips for revealing the information that represent the graph elements. The perfect complement to this usability, is the visual appearance achieved extending the Weka class panel. We consider the outcome a good base to extend the functionality further in next releases of the plug-in.

Having fulfilled the functional and non-functional requirements, we have focused on complementing the package. The entropy triangle is not only a nice visualization, it is a powerful exploratory analysis method [23]. And, we tried to embody all its potential in the plug-in. This is still a work in progress, and as we comment below may involve some interesting design challenges.

The use of the Weka's instances format for the underlying data structure have eased adding file saving and loading to the program. The files can be saved in the Weka's arff format, and exported to csv or json, common file formats that allow us to use the data outside Weka. Each evaluation data is an instance whose attributes are the metrics and identifying data. In csv, for example, a row is an instance, and the columns the attributes.

Although the data structure follows the scheme used by the Weka Experimenter, it is not fully compatible yet. The concrete implementation of format in the plug-in differs in some points from the Experimenter implementation. The Experimenter can produce multiple runs of the same task to bring statistical significance to the evaluation. Although interesting, this ingredient is an issue to handle. We identify here a feature to add in the next version of the plug-in.

5.2 Software distribution

Another important aspect besides developing good software is its distribution. It is necessary to identify the potential community of users in order to make the software available to them. This importance is sometimes overlooked, and there are many good tools that are not used because they are unknown. However, the capacity of developers to promote their programs is often limited.

We uploaded the software to Github[1], a popular repository for open source software. The basic format of this service does not give much information about the program. A random potential user that can be interested in the tools, but not familiarized with Weka or the plug-in content, may discard the package prematurely. With this in mind, we developed a small and informal web site[2]. The content is limited to highlighting the package features, html versions of the manuals, and a programmatic-use example. For the users that decide to use the tools and want a better understanding of them, the main page links to the tool authors' papers.

An interesting characteristic, is that the site is a free service hosted by the same software repository. The use is limited for static html sites, but using Jekyll[3] (a static content generator intended for blogs) we produce them from simple text files in markdown (a syntax for markup language). This scheme facilitates introducing changes to web development agnostics. Moreover, it brings the possibility of introducing examples of use cases as blog posts.

After considering several online services to host the code, we finally decided for Github because, in addition of providing a web site linked to the repository, it has a social network component. In the future, this project may be continued by other students, and the service eases the project management as teamwork. This social component also allows that users provide feedback and bug reports.

5.3 Future work

Open source software development is often collaborative. However, continuing the work done by another person is challenging. We have structured and documented the code trying to ease this task. This document has been written with the same goal. In this section, we try to point out some interesting next steps for this project, from the perspective given by the work already done.

Separate saving and loading buttons

This is the most straightforward extension. The tricky part may be on implementing a sanity check for new loads to be compatible with the instances that the entropy triangle may already be displaying. The current

[1] `https://github.com/apastor/entropy-triangle-weka-package`
[2] `http://apastor.github.io/entropy-triangle-weka-package`
[3] `https://jekyllrb.com/`

implementation is somewhat flexible with attribute names, supplementary restrictions or checks should be added to avoid displaying erroneous information due to a misunderstanding of data.

Experimenter compatibility

We have already argued how interesting this feature is, and how the instance's data format is the key to import the arff files produced by the Experimenter. As the Experimenter is not directly pluggable, this mechanism seems to be the most plausible in order to achieve an indirect compatibility. The issue resides in that the Experimenter can perform multiple runs of the same task for giving the metrics statistical significance.

The Experimenter produces an arff file that stores instances of each run individually. The fields for dataset and classifier match on instances of the same task, and they are differentiated by an attribute for the "run" sequence. The entropy triangle can address these instances in several ways. Two options seem obvious: preprocessing the instances of each task and storing only their mean and standard deviation for all metrics, or storing instances as they come and produce a new instance for the mean value.

The first option reduces memory footprint and can be represented in the plot with markers whose width in each direction of the triangle correspond to the standard deviation. This approach may produce a cleaner diagram than the second option, but the standard deviation of the metric used for the color is not represented. In that case, the tooltips may show the missing value.

The second approach gives the information in raw, producing more populated plots. Although the graph may seem more chaotic, the data can be handled cleanly. A separate dataset of instances can be created for the raw values, and for the means use the same dataset that for regular instances from the Explorer. The `PlotElement` interface feature for handling two data layers may be useful here. The main layer can be drawn like it is done now, and the lower layer used for the cloud of raw instances. A fine finish may be achieved by adding some transparency to the lower layer components to highlight the upper layer data, and setting tooltips only for markers of the main layer.

A comparative table

The authors of the papers describing the tools use a comparative table of some metrics. This can be a nice feature that can be called by a button at the upper panel. The table can be displayed on the entropy triangle window, as a switch of view instead of the ternary diagram, or in a new window. The former implementation keeps the program compact, while the latter allows exploring both sources of data, table and triangle, at the same time.

Different markers for the split view

The split view triples the graph content. The readability of this view can be increased by differentiating the markers of each type of point. The task can be done by creating different classes for each kind of point or adding an attribute to the `DataPoint` class that determines the point shape.

zoom option

Identifying classifiers of points in an overloaded diagram can be annoying. Introducing a zoom option would scale the graph capacity. This can be done by grabbing an area with the mouse, and mapping the ternary plot panel space in screen to that area. The concept can be implemented extending with the `TernaryPlot` the `JScrollPane` class of the JCL. Problems could arise due to the affine transformation, but it should be easily tweaked.

Export the image to eps

The `PrintablePanel` of Weka allows to export the screenshot to several file formats, including the eps format for vectorized graphics. Unfortunately, when this format is chosen to export the entropy triangle, the image shows a window without the ternary diagram and with the color bar in gray scale. This incompatibility may be produced by the plot affine transformation and the linear gradient paint method. Debugging the image saving sequence and reading `PrintablePanel` code could help to find a fix.

Appendices

Appendix A

Installation manual

A.1 Package install

To install the Entropy Triangle package on Weka you can use the Weka Package Manager. If you are in a Unix environment you can do it faster with the command line.

The `EntropyTriangle.zip` package is cross-platform. You can install it on Weka independently of you operating system. The only requirements are that Java is installed (if you already have Weka running this must be the case), and a recent release of the development version of Weka ($\geq$ 3.7.8)

A.1.1 Package Manager GUI

1. In the Weka GUI Chooser, the main Weka window that opens on the program startup, go to the Tools menu and open the Package manager (see Item 1).

Figure A.1: Weka GUI Chooser window

2. On the top-right corner of the Package manager window *click* the **File/URL** button (see Item 2).
3. In the window in Fig. A.3, select the `EntropyTriangle.zip` file with one of the following methods:
   - *Click* the **Browse...** button to select an already downloaded file.
   - Paste in the text box the URL to the package zip file from the release section of the project on github.

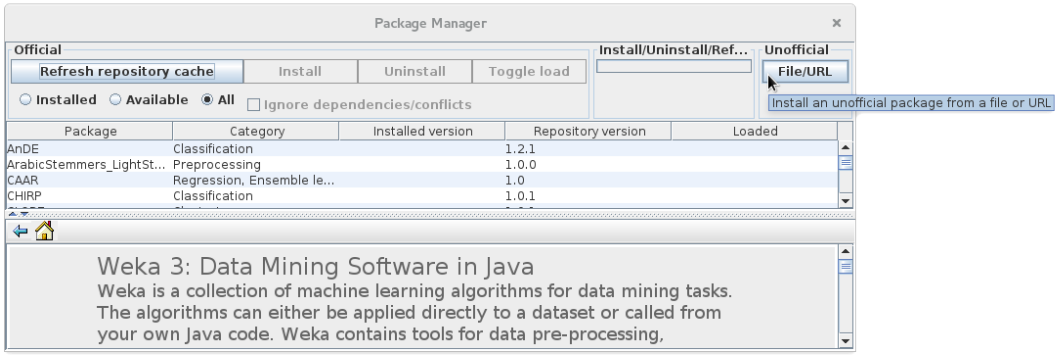


Figure A.2: Weka Package Manager window

Figure A.3: Package file or URL selection dialog

4. **Restart Weka** to get the plugin loaded.

Command line

```
# Add weka.jar to classpath
$ export CLASSPATH=${CLASSPATH}:<path-to>/weka.jar

$ java weka.core.WekaPackageManager -install-package
    ↪ <path-or-url-to>/EntropyTriangle.zip
```

A.1.2 BUILD

The installable zip package is cross-platform, building the package is not necessary. Anyway, to build the package from source you can use Apache Ant.

Run the following command from the project root directory:

```
$ ant -f ./build_package.xml -lib <path-to>/weka.jar make_package
```

You have to specify the project build file with the option `-f` and the weka jar file as library (option `-lib`). Optionally, you can set the build command, `make_package` is the default one.

Appendix B

User manual

The Entropy Triangle package for Weka includes a visualization plug-in and new evaluation metrics. The package can be used from the Weka GUIs and programmatically.

B.1 The Entropy Triangle visualization plug-in

The Entropy Triangle is accessible from the Weka Explorer and the Weka GUI Chooser window.

When called from the Weka GUI Chooser window (Fig. B.1) an empty plot will be displayed. This is the form to open the visualization to load a file with data of another session.

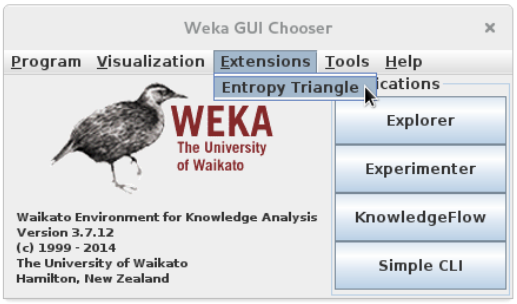


Figure B.1: Weka GUI Chooser window

B.1.1 Adding data from the Explorer

The evaluation data of a classifier tested on the Weka Explorer can be added to the Entropy Triangle.

For that, select on the **Result list** the test you want to add and *right-click* on it to open the context menu. Then, move the cursor to **Plugins** to open the visualization plug-ins sub-menu and click **Entropy Triangle** (seeFig. B.2).

The data will be added to the existing window of the Entropy Triangle. If you do not have the Entropy Triangle window already opened, a clean new plot will be generated.

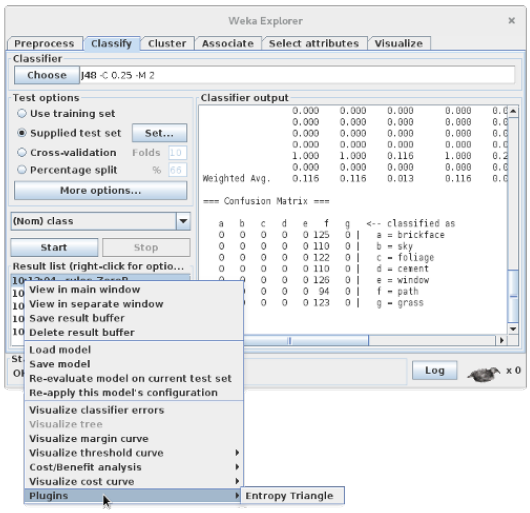


Figure B.2: Weka Explorer, Classify tab. Plug-ins menu in the context menu of the result list

B.1.2 Changing the colorbar metric

The bottom panel of the Entropy Triangle titled **Metric color** is a modified version of the Weka's "Class color" panel.

The metric can be changed *clicking* in the upper-left combo box. This will open a drop-down list with all the available metrics and related information about the data. Choosing an evaluation metric draws a colorbar with the values adjusted to the range of the metric. When the label represents a non numeric value, i.e. the relation name or the classifier, colored text labels are drawn. As in the Weka class panel, the color of the text labels can be changed by *clicking* on them.

B.1.3 Split mode

The **Split** button toggles between showing in the plot the points that represent the marginals entropy balance equations. For more information about this, see the Section 2.1.2.

B.1.4 ΔH'(X) line

The ΔH'(X) line marks the normalized distance in entropy of the dataset likelihood from the maximum. This distance determines the upper bound of the mutual information $MI' \leq 1 - \Delta H'_X$. Recall that maximum entropy entails a balanced dataset, and this is the only case where the normalized mutual information can be maximal $MI' = 1$ for the best possible EMA.

This line defines the limit of performance of the classifier due to the test set.

B.1.5 Removing data

To remove a point from the plot, *right-click* on it to open the point context menu, then select **delete**.

To erase all the data, close the window. A clean empty plot will be loaded when opening again the Entropy Triangle window or adding data.

B.1.6 Saving to a file

The Entropy Triangle lets you save the plotted data. The information is saved in the Weka instances format, in a similar way to Weka Experimenter files. Each classifier tested is an instance, and their information are the attributes. This is the dataset name, classifier name and options, evaluation metrics, and timestamp information.

When the Entropy Triangle has data plotted, the upper-right button is titled **Save data**. *Click* on it to open a dialog to choose the file name and format. You can choose between several file formats, the native Weka file formats: arff, xrff and the binary bsi, and CSV or JSON, to export the data more easily.

B.1.7 Loading a file

Open the Entropy Triangle from the Weka GUI Chooser window "Extensions" menu. The upper-right button of the empty plot will be named **Load data**. *Click* on it to open the file selection dialog.

B.1.8 Taking a screenshot the plot

To take a screenshot of the graph window do *Ctrl+Shft+Alt+Left Mouse Click*. You can save the image in bmp, jpg or png.

B.2 Package Metrics

The EMA and the NIT factor will appear on the evaluation report without any additional user action, as they are implemented as standard evaluation metrics.

B.2.1 How to output information-theoretic metrics

The rest of the package metrics are implemented as information-theoretic evaluation metrics. This means that are off by default, and only added to the evaluation report by user request.

To ouput them in the Weka Explorer go to the **Classify** tab, open the **More options...** window, and set the **Output entropy evaluation measures** option. If you want to print metrics selectively, open the **Evaluation metrics...** dialog and unselect those that you want to remove from the report.

In the Weka Experimenter all the metrics are available by default.

Appendix C

Legal framework

The information about the rights and permissions of the licenses mentioned in this section is provided as a summary and does not substitute the licenses content.

C.1 Software

The software implemented in this project is licensed under the GNU General Public License (GPL) version 3[1]. A copy of the license is provided with the official package distribution at Github[2] in the plain text file "LICENSE.txt".

The GNU GPL v3 license provides no warranty of any kind for the software [11]. Redistribution of the software is allowed within the terms of the license:

- The copies should contain an appropriate copyright notice.
- Keep intact all notice stating the software license.

The software can be modified. The author encourages anyone to modify the software freely. Modified versions of the program can be redistributed with the following terms:

- There must be clear that you modify the program and include a relevant date.
- Previous copyright notices should be maintained.
- Any software derived must be licensed in its entirety under the terms of the GNU GPL license v3, or optionally any newer version.

C.2 Document

This document is licensed under the terms of the Creative Commons Attribution-NonCommercial 4.0 International license[3] (CC BY-NC 4.0). Anyone is allowed to share and adapt this document under the following terms [4]:

[1] http://www.gnu.org/licenses/gpl-3.0.en.html
[2] https://github.com/apastor/entropy-triangle-weka-package
[3] http://creativecommons.org/licenses/by-nc/4.0/

- Provide attribution, indicating any change.
- The document can not be used for commercial purposes.
- There can not be applied additional restrictions to the terms of the license.

Appendix D

Budget

The project has been realized with academical and non-profit purposes. However, the time and material dedicated to it has a non negligible cost. Obtaining funding for this type of projects would help significantly to extend the knowledge of scientific advances. We provide an accurate estimation of the cost associated to the project with the intention of encouraging such funding for future updates or improvement proposals.

The Table D.1 shows the project stages and their estimated dedication time by the student. The tutor dedication is approximated to a 30% of the student time, to tutor and correct the work.

Table D.1: *Project stages*

Stage	Duration
Documentation	50 *hours*
Software development	100 *hours*
Software testing	50 *hours*
Project web page	20 *hours*
Project report	100 *hours*

We consider a stipend of 10 €/hour for the student and 60 €/hour for the tutors. The personnel cost is depicted in the Table D.2

Table D.2: *Personnel costs*

Expenditure	Unit price	Amount
Student	10 €/hour	3200 €
Tutor	60 €/hour	5760 €
	Total	**8.960 €**

The material costs, disaggregated in the Table D.3, consider a duration time for the project of 6 months. The equipment used by the student consists on a laptop and a desk at a shared office. The student laptop only uses free software and has a price of 600€ with a lifetime of 2 years, representing a cost of 25 €/month. The desk rental includes an internet connection. The tutor material cost is included in his/her stipend of 60€/hour. And finally, we consider an entry grouping other expenses, like stationery bills.

Table D.3: *Material costs*

Expenditure	**Unit price**	**Amount**
Laptop	25 €/month	150 €
Desk	150 €/month	900 €
Other expenses	20 €/month	120 €
	Total	**1.170 €**

All things considered, the Table D.4 present the project budget.

Table D.4: *Budget*

Expenditure	**Amount**	
Personnel costs	8.960	€
Material costs	1.170	€
Taxable	10.130	€
VAT (21%)	2.127,30 €	
TOTAL	**12.257,30 €**	

Appendix E

Planning

The project can be clearly divided in five stages: documentation, software development, software testing, web site elaboration, and the elaboration of this document. The Table E.1 shows their estimated duration.

Table E.1: *Project stages*

Stage	**Duration**
Documentation	50 *hours*
Software development	100 *hours*
Software testing	50 *hours*
Project web site	20 *hours*
Report elaboration	100 *hours*

The project will be performed part-time in six months. Then, the 320 hours can be spread over 25 weeks. To finalize on time, the average required dedication is 12,8 hours/week. We are going to be a bit cautious and dedicate 15 hours weekly. Thus we will have more than 3 weeks of margin for setbacks.

The software development can be divided into design and implementation. The design can overlap the documentation in period. Design drafts help identify crucial parts of the program. Thereby, they can drive the scope of the documentation after an initial period. Similarly, the development of the software is slightly overlapped with the testing stage, as verified functionality helps implementing components that add up to it. The final part of the testing stage is intended to develop use cases for the software. We will use them in the web and project report. Finally, the web site and the report can be elaborated in parallel after having closed a working version of the program that satisfies the requirements.

Fig. E.1 illustrates the distribution of the stages in a Gantt diagram. The critical task is to achieve a working implementation that meets the requirements in time, since delaying this milestone would force to shift subsequent tasks.

Name	Work
Documentation	30d
Development	**35d**
Design	10d
Implementation	20d
Merging	5d
Testing	**20d**
functionality testing	10d
Use cases	10d
Web site	10d
Report elaboration	35d

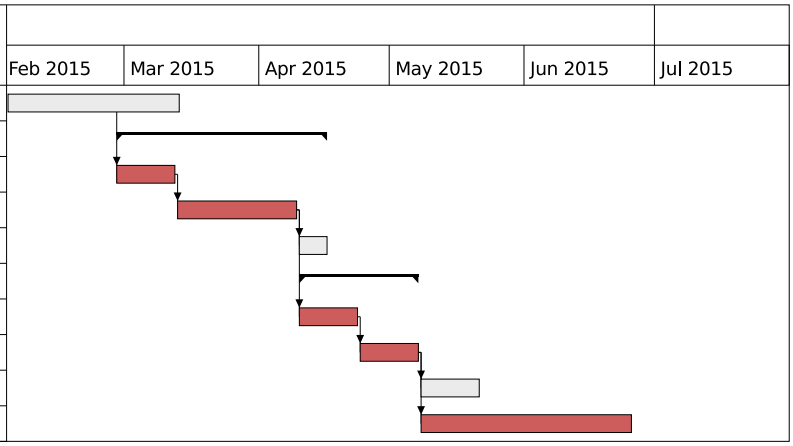


Figure E.1: Gantt diagram

Appendix F

Extended abstract

This document presents a software plug-in to endow the Weka machine learning suite [8] with a set of information-theoretic tools for the assessment of classifiers [22, 23]. The utility of these tools is more evident in multi-class classification, but they can be used as well for binary tasks.

In this extended abstract we overview the main parts of the document. We outline the implemented tools, the Weka software, and the plugin design. The main part presents an use case for a multi-class dataset in which we unbalance the class distribution in different ways. And finally, we review the project in hindsight.

F.1 Theoretical background

The use of information theory to assess classifiers is grounded on the analogy between a classification process and a communication channel [16]. In this context, the input and the output of the classifier can be viewed as discrete random variables X and Y, respectively, with k—the number of classes—different possible values. Their marginal and joint probabilities can be estimated from the confusion matrix of the classifier testing, which tallies the label class of the classified instances against their predicted class. Then, the maximum likelihood estimate for the joint probability of every label-prediction pair is the element of the matrix at such index divided by the total number of instances of the test set

$$\hat{P}_{XY} = C_{XY}/N$$

Although the confusion matrix contains complete information to evaluate the classifier, its direct inspection, e.g. as a *heatmap*, is not practical, specially in the multi-class classification problems. Information-theoretic tools help us to extract the information from the confusion matrix in a more apprehensive form.

The Entropy Triangle

The Entropy Triangle represents in a De Finetti diagram, or ternary plot, a normalized balance equation of entropies for the estimated distributions of the input and the output

$$1 = \Delta H'_{P_X \cdot P_Y} + 2MI'_{XY} + VI'_{XY} \tag{F.1}$$

The maximum value of a entropy correspond to the uniform distribution, where all the values are equally likely. Therefore, the upper bound for the joint entropy of two variables is the case when they both are

uniform and independently distributed. The terms of the right side of Eq. (F.1) are normalized by this upper bound.

The mutual information measures the amount of information that these variables share. The variation of information is the opposite concept to the mutual information and can be interpreted as the distance between the two random variables [15]. And $\Delta H_{P_X \cdot P_Y}$ is the difference in the joint entropy of the variables from the upper bound. Fig. F.1 shows the Entropy Triangle identifying the performance qualities for classifiers located at the vertices and sides.

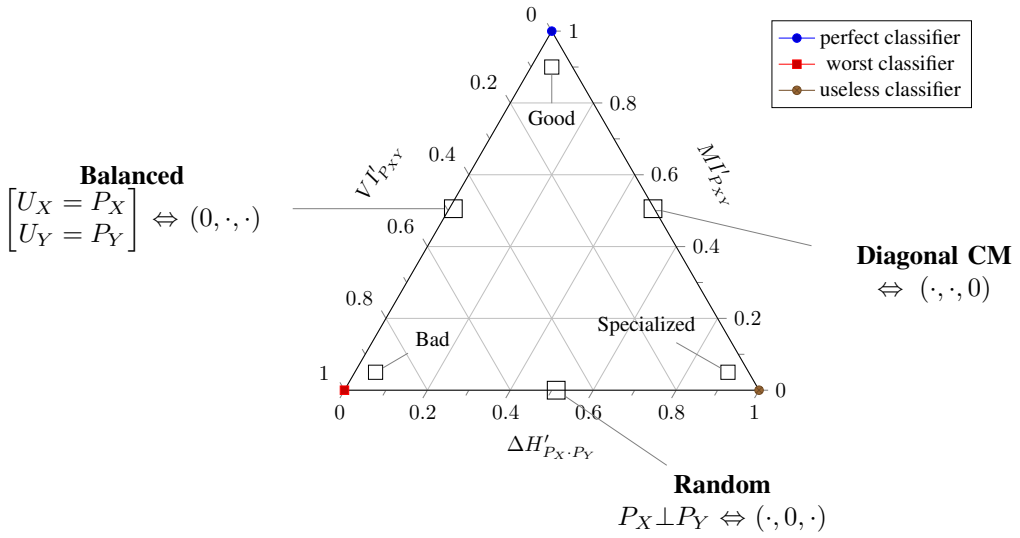


Figure F.1: Entropy Triangle showing interpretable zones and extreme cases of classifiers (reprinted from [22])

This diagram provides, at a glance, complete information of the confusion matrix in terms of information theory. Multi-class classification tasks lacked of good visual characterizations, like the *Receiver Operating Characteristic* (ROC) curve [6] for binary classification. For this reason, the Entropy Triangle appears as a powerful tool for the reliable assessment of multi-class classifiers [23].

Split view of the marginal distributions

In addition, the Entropy Triangle can represent separate balance equations for the marginal distributions. The equations are now normalized by H_{U_X} or H_{U_Y}, respectively

$$1 = \Delta H'_{P_X} + MI'_{XY} + VI'_X \tag{F.2a}$$

$$1 = \Delta H'_{P_Y} + MI'_{XY} + VI'_Y \tag{F.2b}$$

where $VI'_X = H'_{P_{X|Y}}$ and $VI'_Y = H'_{P_{Y|X}}$.

Upper bound to the mutual information due to dataset likelihoods

A desirable feature for a classifier is to maximize the mutual information between the input and the output. But, the mutual information is conditioned by the equation balance.

To illustrate, suppose a perfect classifier that transfers all the information it has to the output, leaving no $H_{P_{X|Y}} = 0$. Then, Eq. (F.2a) becomes $1 = \Delta H'_{P_X} + MI'_{XY}$. As P_X does not depend of the classifier, this is an upper bound for the mutual information

$$MI'_{XY} \leq 1 - \Delta H'_{P_X} \tag{F.3}$$

We can draw a line in the Entropy Triangle for $\Delta H'_{P_X}$, the normalized distance in entropy of the dataset likelihood from the maximum. This line draws an intuition of the limit of performance of the classifier due the test set.

Perplexity-based metrics for classifiers

Besides the Entropy Triangle, we implement in the package some useful metrics for the assessment of classifiers based on the perplexity. The perplexity is an information-theoretic measure of difficulty used to indicate the number of possible values for discrete variables [1]. In our context, the perplexity represents the effective number of classes for the classification task [22]. The perplexity PP of a discrete variable is calculated in terms of its entropy

$$PP_X = 2^{H_X}$$

From the relation of the accuracy with the perplexity, the **Entropy Modified Accuracy** is specified in terms of perplexity as a pessimistic estimate of the accuracy [22]. The entropy modified accuracy is defined as the inverse of the remaining perplexity

$$EMA = \frac{1}{PP_{P_{X|Y}}} = \frac{1}{2^{H_{X|Y}}} = \frac{1}{k_{X|Y}} \tag{F.4}$$

where $1/k_X \leq EMA \leq 1$. Here, we call k_X to the perplexity of the input class.

The **Normalized Information Transfer factor** is defined, as its name suggests, like the information transfer factor, normalized by the number of classes [22]

$$NITfactor = \frac{\mu_{XY}}{k} = \frac{2^{MI_{XY}}}{k} \tag{F.5}$$

we can relate the NIT factor to the EMA

$$NITfactor = \frac{\mu_{XY}}{k} = \frac{k_X}{k}\frac{1}{k_{X|Y}} = \frac{k_x}{k}\ EMA \tag{F.6}$$

where $k/k_X = 2^{H_{U_X} - H_{P_X}} = 2^{\Delta H_{P_X}}$, from Eq. (2.8) in terms of perplexities.

The relation of Eq. (F.6) between the EMA and the NIT factor also bind their bounds

$$\frac{1}{k} \leq NITfactor \leq EMA \leq 1$$

Where the EMA only can equal the lower bound $1/k$ and the NIT factor the upper bound 1 when the likelihoods are uniform. This limitations do not suppose a practical problem, but suggest different uses for both metrics. In [22], the EMA is recommended to rank classifiers, and the NIT factor to measure the classifiers level of understanding of the underlying patterns of the task.

F.2 The Weka software

The *Waikato Environment for Knowledge Analysis* (WEKA) [8] is a workbench for machine learning and data mining developed at the University of Waikato, New Zealand. The project has more than twenty years of history [9].

Weka has different Graphical User Interfaces (GUIs) available, that let the user choose from an user friendly interactive explorer, to an automated approach where multiple experiments can be statistically compared at the same time. Advanced users can use the program from the command line or *programmatically* in code, a mechanism available for Java, C, Python, etc. Accordingly, it includes an extensive list of algorithms, data processing tools, and visualization facilities. An important feature of Weka is the possibility to use it as a framework for the implementation of algorithms, evaluation metrics and visualization tools by means of added components. The current version of Weka (3.7) has a package manager that eases installation and sharing of custom components as *plug-in packages*.

The program is implemented in Java, so the Java object-oriented architecture is a convenient environment to develop our tool packages. In addition, since we have to implement a visualization tool, the Java built-in graphic libraries are of great help. For example, making the graph elements interactive.

Interfacing Weka

Weka has an object oriented architecture and is implemented in the Java programming language. As it is natural in Java, Weka uses inheritance to extend functionality and interfaces to interact with plug-ins.

The metrics classes must extend the `weka.classifiers.evaluation.AbstractEvaluationMetric` abstract class and implement one of the available interfaces, attending to the type of metric. The plug-in manager handles the addition of new evaluation metrics to Weka, and makes them available in all the Weka user interfaces. For that. the metrics have to be listed in a configuration file at the package root directory named `PluginManager.props` [32].

The Entropy Triangle is plugged through the Classify panel of the Weka Explorer for adding data of the performed tasks, and at the GUI Chooser window that is loaded on Weka startup for loading data from files. In the Classify panel the plug-ins are available through the right-click menu of the result history list. This sub menu, will show elements that fit the next two conditions:

- Reside in the package `weka.gui.visualize.plugins`
- Implement one of the interfaces of the package.

The GUI Chooser window has a pluggable "Extensions" menu [31] that adds items with two requirements:

- The class of the menu component has to implement the `weka.gui.MainMenuExtension` interface.
- The `GenericPropertiesCreator.props` configuration file has to list the package of the class under the `weka.gui.MainMenuExtension` entry. This way, the automatic class discovery feature of Weka will find our class implementing the aforementioned interface. We have to create this file in the plug-in root directory.

The plug-in packages for Weka have to be packed in a zip file and follow a particular directory structure described at [32].

F.3 System Design

The design stems from the Weka interfacing options and their requirements. From this point, we have analyzed different options for the components of the plug-in, and tried to choose those that fitted best with the global design and individual tasks.

Overall, we have pursued an object-oriented hierarchical and modular design. For that, we used inheritance for a clean and highly abstracted implementation, and a hierarchical organization of components for a better isolation of scopes. Other essential considerations include building a program as robust and lightweight as possible. We also took advantage of the potential offered by Weka and Java for several tasks.

The design of the package has been planned in advance, although the implementation of the software classes and functions has been done incrementally. We implemented the evaluation metrics in the first place as they are needed for the Entropy Triangle visualization. Then, we continued with a basic ternary plot making it callable from Weka. Then, we gradually added functionality until reaching the proposed objectives. We think that this approach has driven us to a better final design because the project was implemented from the basic functions, but having a concept of the entire goal from the beginning. This way we could evaluate the impact of each step, and consider redesigns when appropriate.

Package structure

The Java classes of the plug-in are grouped in two Java packages: `weka.etplugin` for the Entropy Triangle elements and panels, and `weka.etplugin.metrics` for the evaluation metrics. The only exception is the `EntropyTrianglePlugin` class, that handles the calls from Weka to the Entropy Triangle. This class, as explained in Appendix F.2, has to belong to `weka.gui.visualize.plugins` package. Figure F.2 shows a class diagram that outlines the `weka.etplugin` package.

Classes and methods

The main class of the plug-in is the `EntropyTrianglePanel`. This panel acts as coordinator of the program flow and top level container of buttons, ternary plot and color bar panels.

The plot is divided in different Java classes where each one represents a graph element. The `TernaryPlot` class is the Java panel that contains the plot elements as children. The elements handle their own functionality, like the response to mouse events for showing the tooltips and in the case of the `Baseline` opening the dialog for changing its color. The `TernaryPlot` panel use the `AffineTransform` class of the Java package `java.awt.geom` to provide a graph coordinate system that simplifies the design.

For the `ColorBar` panel we modified the `ClassPanel` class of Weka to add the heatmap effect of a rainbow style gradient. The `LinearGradientPaint` class of the `java.awt` package is used for painting the gradient. The gradient colors are stored in a data buffer and used by the plot elements for they foreground color. A method maps the buffer indices to metrics values.

The scheme of the pluggable evaluation metrics is constrained by the Weka interfacing design. The entropies are computed via the static methods of the Weka class `weka.core.ContingencyTables`, and the perplexities with the `Math` class of the Java standard libraries.

Data structure

A backbone of objects interaction is the main data structure. The instances format of Weka provided us with the possibility to outsource all the data handling with the Weka API. This format implements hash tables, that apart from being memory-efficient are also computationally efficient.

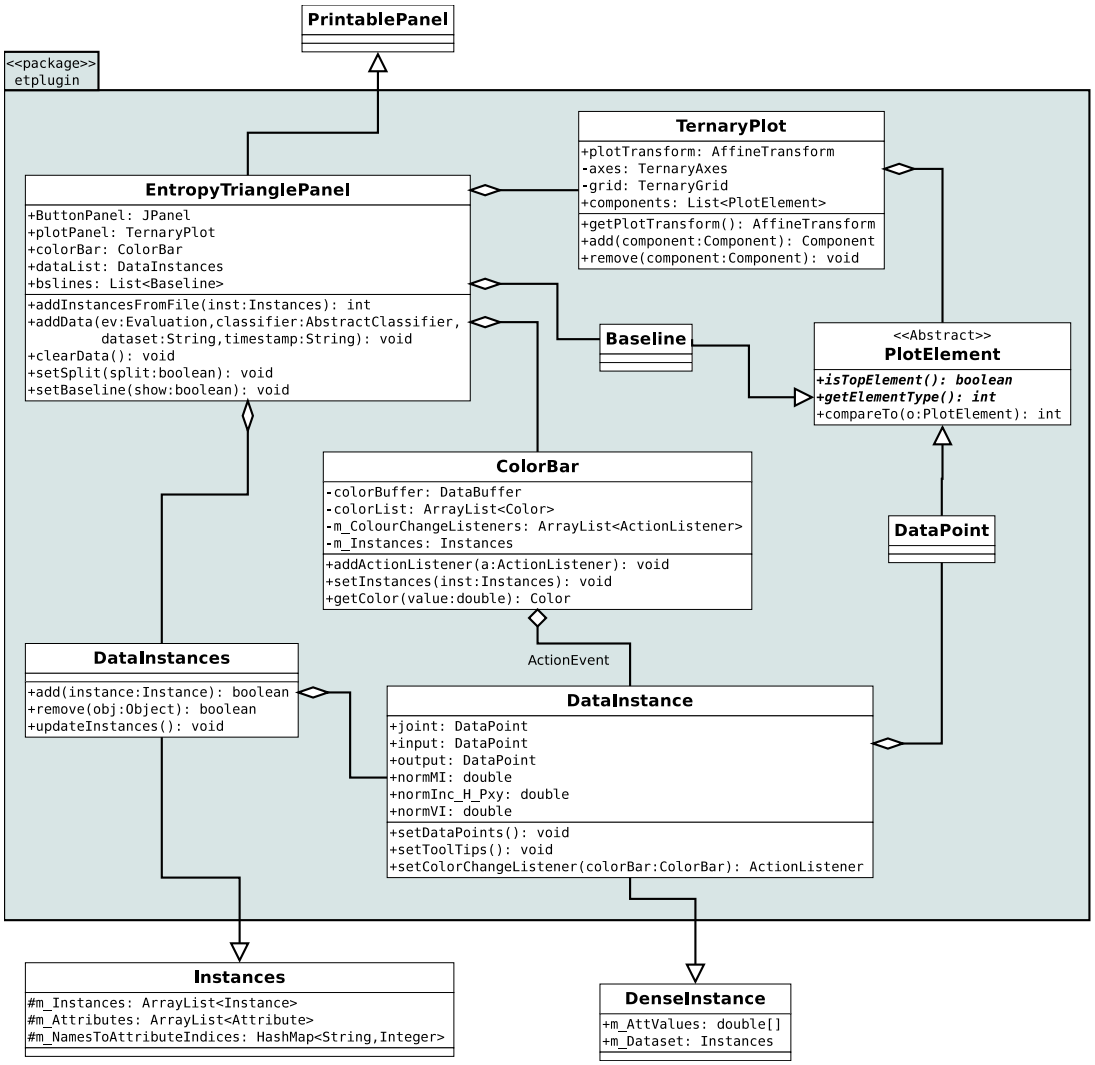


Figure F.2: `etplugin` package class diagram

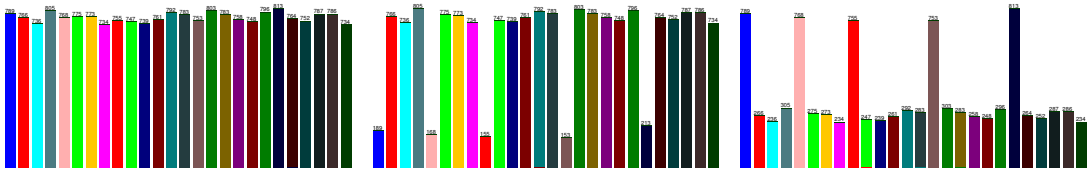
We extended the `Instance` class of Weka with the `DataInstance` class, inheriting the efficient structures and incorporating the attributes we needed for the Entropy Triangle.

In the background, setting the class attribute for the metric chosen to colorize the graph points simplifies the interaction. `DataInstance` objects create action listeners that the `EntropyTrianglePanel` binds to `ColorBar` events.

F.4 Use case

In this section we show how the Entropy Triangle works. To evidence the effect of unbalanced datasets on the metric values, we created two subsets with unbalanced class distribution from the Letter dataset, 26 classes corresponding to the letters of the alphabet [7]. We removed 600 instances of each vowel (Fig. F.3(b)) in one of them, and 500 instances of each consonant (Fig. F.3(c)) in the other.

For this example we use the following classifiers: zero rules, logistic regression [14], naive Bayes [12], and C4.5 [19] (J48 in Weka). The classifiers are trained with the 70% of instances and evaluated with the

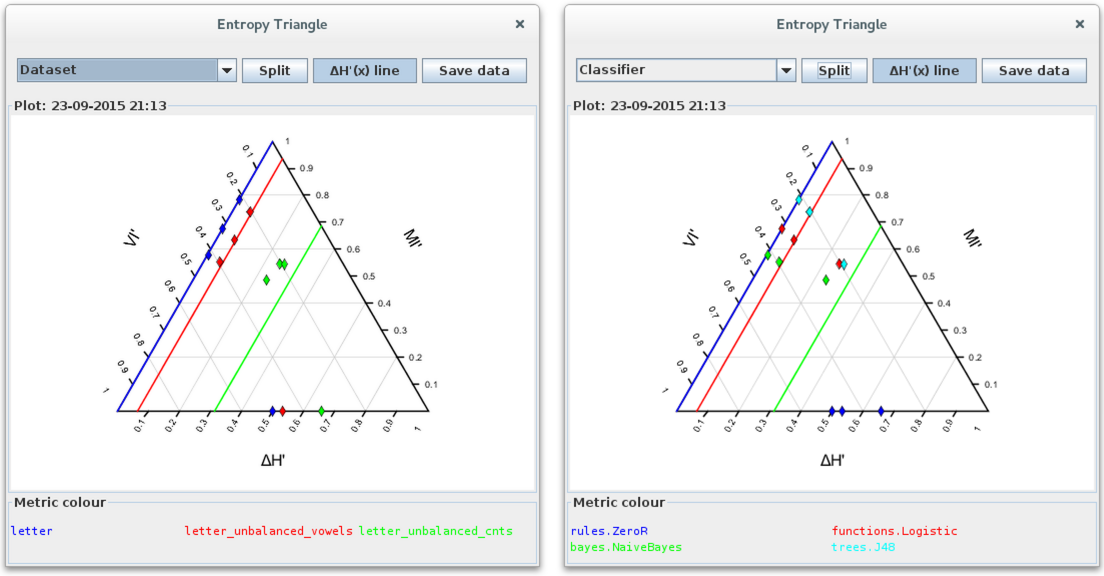


(a) Complete dataset (b) Removed vowels dataset (c) Removed consonants dataset

Figure F.3: Letter dataset class distribution

remaining 30%, all with their default parameters.

The data in the Entropy Triangle can be identified interactively by means of the tooltips on mouse hover. Besides, we can have an overview selecting the dataset or classifier in the combo box to colorize the data, like in Fig. F.4.



(a) Data colorized by dataset (b) Data colorized by classifier

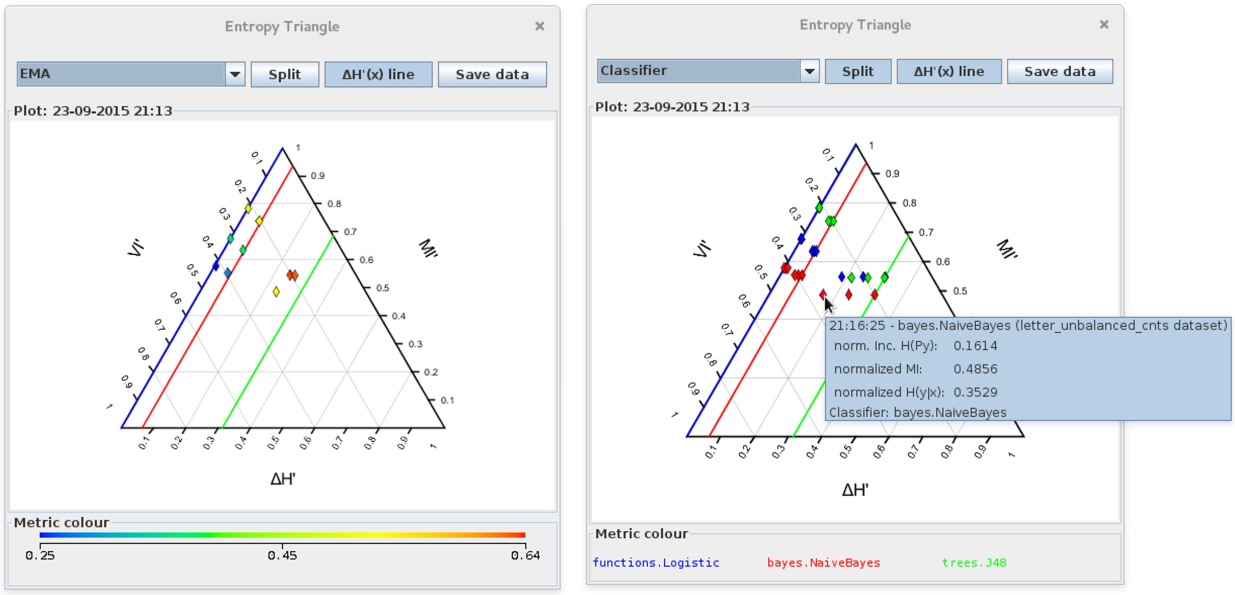
Figure F.4: Entropy Triangle window with different classifiers for the unbalanced letter datasets

We can appreciate that C4.5 seems to outperform the other classifiers for the balanced dataset and the one with removed vowel instances. In the dataset with removed consonants the C4.5 and logistic classifiers seems to have equal mutual information (MI). They difference in terms of variation of information (VI) or increment of entropy from the uniform distribution (ΔH) does not seem to be significant as well. The zero rule classifier shows a bad performance for the three datasets. This happens as expected because none of the sets has a single majority class.

Using the evaluation metrics and the split view

To characterize the performance of the classifiers through evaluation metrics we remove the zero rule results from the Entropy Triangle. This way, the colorbar range is shorten and differences in closer values are better

appreciated. Fig. F.5 shows the data colorized by the entropy modulated accuracy (EMA) and in split view by classifier.



(a) Data colorized by EMA

(b) Spliw view colorized by classifier

Figure F.5: Entropy Triangle windows

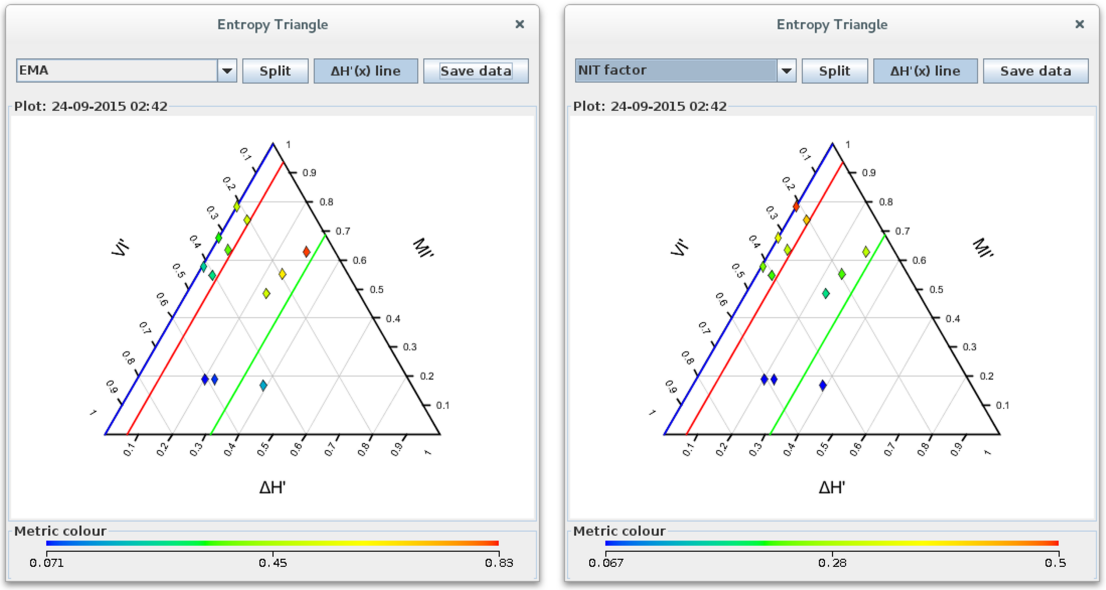
We can see that the EMA values for the logistic and C4.5 classifiers in the removed consonants dataset are higher than in the rest of cases. This value is explained because EMA is the inverse of the remaining perplexity, and for this dataset the classifiers have achieved a higher reduction of perplexity than for the other more balanced datasets. The split mode of the Entropy Triangle shows also this improvement with the P_Y markers of these classifiers on the left of the P_X ones (Fig. F.5(b)).

Unbalanced evaluation of balanced trained classifiers

The maximum values of the mutual information are limited by the class distribution of the test sets. We identify this limit in the Entropy Triangle with the ΔH_{P_X} lines. To analyze this constraint we performed evaluations with the unbalanced sets of the classifiers trained with the balanced set. We changed the baseline from the zero rule classifier to the one rule classifier [10].

Fig. F.6 shows the results, that are quite similar to the case with unbalanced training. The only classifier that improves is the C4.5 for the removed constants dataset, showing that with a balanced training can predict better. Anyway, the values are limited by the class distribution of the test set.

These results give an insight of how to interpret the values in the Entropy Triangle. Performance of classifiers is determined by their training data and learning scheme, but the testing conditions may influence their scores. Therefore, there are necessary tools for a reliable assessment, independently of the data available for evaluation. The Entropy Triangle and the metrics of the package provides an information-theoretic framework for the assessment of classifiers that take into account the context in which they are evaluated.



(a) Data colorized by EMA

(b) Data colorized by NIT factor

Figure F.6: Entropy Triangle with unbalanced evaluation of balanced trained classifiers

F.5 Conclusions

There are projects that can extend for long in time. Software developments are often affected by that: there are always things that can be either improved or added. Therefore, it is wise to split them in stages. The Entropy Triangle Package for Weka is one of that kind of project. In this thesis we have documented the first phase. This section describes the conclusions we have drawn from the project. Besides evaluating the work of this months, we outline what we consider the natural next steps.

Software development

We proposed to implement the complete set of information-theoretic tools described in [22, 23] as a plug-in for the Weka machine learning suite. The first release of the program that we present here meets this main requirements successfully. We have produced a working program that can be easily installed through the Weka standard procedure for packages. Moreover, it is accessible while unobtrusive.

The architecture designed for the package has a high modularity and a hierarchical structure. The Java classes represent concrete elements of the program and we have abstracted common functionality to base objects. Similarly, we interfaced and extended features available in Weka and the Java standard libraries to produce better software with less complexity and code, like extending the Weka `PrintablePanel` for printing the window to an image. An important aspect, mathematically speaking, is that we have outsourced the complicated calculus to efficient and tested classes of Weka and Java. As a result, we have a more robust program.

From the user perspective, the Entropy Triangle is highly interactive. The graph is kept tidy but contains complete information. There are two key components in this result: the button bar to manage the graph content, and the use of tooltips for revealing the information that represent the graph elements. We consider the outcome a good base to extend the functionality further in next releases of the plug-in.

Having fulfilled the functional and non-functional requirements, we have focused on complementing the

package. The Entropy Triangle is not only a nice visualization, it is a powerful exploratory analysis method [23]. And, we tried to embody all its potential in the plug-in. This is still a work in progress, and as we comment below may involve some interesting design challenges.

The use of the Weka's instances format for the underlying data structure have eased adding file saving and loading to the program. The files can be saved in the Weka's arff format, and exported to csv or json, common file formats that allow us to use the data outside Weka.

Although the data structure follows the scheme used by the Weka Experimenter, it is not fully compatible yet. The concrete implementation of format in the plug-in differs in some points from the Experimenter implementation. The Experimenter can produce multiple runs of the same task to bring statistical significance to the evaluation. Although interesting, this ingredient is an issue to handle. We identify here a feature to add in the next version of the plug-in.

Open source software development is often collaborative. However, continuing the work done by another person is challenging. We have structured and documented the code trying to ease this task. This document has been written with the same goal. In Section 5.3, we try to point out some interesting next steps for this project, from the perspective given by the work already done.

Software distribution

Another important aspect besides developing good software is its distribution. It is necessary to identify the potential community of users in order to make the software available to them. This importance is sometimes overlooked, and there are many good tools that are not used because they are unknown. However, the capacity of developers to promote their programs is often limited.

We uploaded the software to Github[1], a popular repository for open source software. With the intention to attract potential users, we developed an small and informal web site[2]. The content is limited to highlighting the package features, html versions of the manuals, and a programmatic-use example. For the users that decide to use the tools and want a better understanding of them, the main page links to the tool authors' papers.

In the future, this project may be continued by other students, and the online hosting service eases the project management as teamwork. This social component also allows that users provide feedback and bug reports. The web can be expanded introducing examples of use cases as blog posts.

[1] `https://github.com/apastor/entropy-triangle-weka-package`
[2] `http://apastor.github.io/entropy-triangle-weka-package`

Bibliography

[1] Lalit R Bahl, Frederick Jelinek, and Robert L Mercer. "A maximum likelihood approach to continuous speech recognition." In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 2 (1983), pp. 179–190 (cit. on pp. 7, 51).

[2] Arie Ben-David. "A lot of randomness is hiding in accuracy." In: *Engineering Applications of Artificial Intelligence* 20.7 (2007), pp. 875–885 (cit. on p. 8).

[3] Remco R. Bouckaert et al. *WEKA Manual for version 3-7-12*. Dec. 16, 2014 (cit. on pp. 12, 15).

[4] Creative Commons. *Attribution-NonCommercial 4.0 International license*. URL: `http://creativecommons.org/licenses/by-nc/4.0/legalcode` (visited on 09/01/2015) (cit. on p. 43).

[5] Robert M. Fano. *Transmission of Information: A Statistical Theory of Communication*. The MIT Press, 1961, p. 400 (cit. on p. 4).

[6] Tom Fawcett. "An introduction to ROC analysis." In: *Pattern Recognition Letters* 27.8 (2006), pp. 861–874 (cit. on pp. 6, 50).

[7] Peter W. Frey and David J. Slate. "Letter recognition using Holland-style adaptive classifiers." In: *Machine learning* 6.2 (1991), pp. 161–182 (cit. on pp. 25, 54).

[8] Mark Hall et al. "The WEKA data mining software: an update." In: *SIGKDD Explorations* 11.1 (2009), pp. 10–18 (cit. on pp. 1, 9, 49, 52).

[9] Geoffrey Holmes, Andrew Donkin, and Ian H. Witten. "Weka: A machine learning workbench." In: *Intelligent Information Systems, 1994. Proceedings of the 1994 Second Australian and New Zealand Conference on*. IEEE. 1994, pp. 357–361 (cit. on pp. 9, 52).

[10] Robert C Holte. "Very simple classification rules perform well on most commonly used datasets." In: *Machine learning* 11.1 (1993), pp. 63–90 (cit. on pp. 26, 56).

[11] Free Software Foundation Inc. *GNU GENERAL PUBLIC LICENSE version 3*. URL: `http://www.gnu.org/licenses/gpl-3.0.en.html` (visited on 09/01/2015) (cit. on pp. 24, 43).

[12] George H. John and Pat. Langley. "Estimating continuous distributions in Bayesian classifiers." In: *Proceedings of the Eleventh conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc. 1995, pp. 338–345 (cit. on pp. 25, 54).

[13] Jonathan Knudsen. *Java 2D graphics*. " O'Reilly Media, Inc.", 1999 (cit. on p. 18).

[14] Saskia Le Cessie and Johannes C. Van Houwelingen. "Ridge estimators in logistic regression." In: *Applied statistics* (1992), pp. 191–201 (cit. on pp. 25, 54).

[15] Marina Meila. "Comparing clusterings—an information based distance." In: *Journal of Multivariate Analysis* 28 (2007), pp. 875–893 (cit. on pp. 4, 50).

[16] George A. Miller and Patricia E. Nicely. "An analysis of perceptual confusions among some English consonants." In: *The Journal of the Acoustical Society of America* 27.2 (1955), pp. 338–352 (cit. on pp. 3, 49).

[17] Terry Ngo. "Data Mining: Practical Machine Learning Tools and Technique, Third Edition by Ian H. Witten, Eibe Frank, Mark A. Hell." In: *SIGSOFT Softw. Eng. Notes* 36.5 (Sept. 2011), pp. 51–52 (cit. on p. 9).

[18] Gregory Piatetsky. *KDnuggets 15th Annual Analytics, Data Mining, Data Science Software Poll: RapidMiner Continues To Lead*. June 2014. URL: http://www.kdnuggets.com/2014/06/kdnuggets-annual-software-poll-rapidminer-continues-lead.html (cit. on p. 9).

[19] John Ross Quinlan. *C4. 5: Programs for Machine Learning*. Vol. 1. Morgan Kaufmann, 1993 (cit. on pp. 25, 54).

[20] Herbert Schildt. *Java: A Beginner's Guide*. 6th. McGraw-Hill Osborne Media, 2014 (cit. on pp. 15, 23).

[21] Claude E. Shannon. "A mathematical theory of communication." In: *Bell System Technical Journal, The* 27.3 (July 1948), pp. 379–423 (cit. on p. 3).

[22] Francisco J. Valverde-Albacete and Carmen Peláez-Moreno. "100% Classification Accuracy Considered Harmful: The Normalized Information Transfer Factor Explains the Accuracy Paradox." In: *PLoS ONE* 9.1 (Jan. 2014), e84217 (cit. on pp. 1, 3, 5–10, 33, 49–51, 57).

[23] Francisco J. Valverde-Albacete and Carmen Peláez-Moreno. "Two information-theoretic tools to assess the performance of multi-class classifiers." In: *Pattern Recognition Letters* 31.12 (2010), pp. 1665–1671 (cit. on pp. 1, 3, 6, 9, 10, 33, 49, 50, 57, 58).

[24] Eric W. Weisstein. *"Affine Transformation." From MathWorld–A Wolfram Web Resource*. URL: http://mathworld.wolfram.com/AffineTransformation.html (visited on 09/01/2015) (cit. on p. 18).

[25] *Weka API, development version*. URL: http://weka.sourceforge.net/doc.dev/ (visited on 05/13/2015) (cit. on pp. 10, 12, 23).

[26] *Weka homepage*. URL: http://www.cs.waikato.ac.nz/ml/weka/ (visited on 05/11/2015) (cit. on p. 9).

[27] *Weka homepage, development info*. URL: http://www.cs.waikato.ac.nz/ml/weka/development.html (visited on 05/11/2015) (cit. on p. 12).

[28] *Weka wiki*. URL: http://weka.wikispaces.com/ (visited on 05/13/2015) (cit. on pp. 10, 12).

[29] *Weka wiki, `build_package.xml` template*. URL: http://weka.wikispaces.com/file/detail/build_package.xml (visited on 06/05/2015) (cit. on p. 15).

[30] *Weka wiki. `Description.props` template*. URL: http://weka.wikispaces.com/file/detail/Description.props (visited on 06/05/2015) (cit. on p. 15).

[31] *Weka wiki. Extensions for Weka's main GUI*. URL: http://weka.wikispaces.com/Extensions+for+Weka's+main+GUI (visited on 05/27/2015) (cit. on pp. 14, 52).

[32] *Weka wiki. How are packages structured for the package management system?* URL: http://weka.wikispaces.com/How+are+packages+structured+for+the+package+management+system? (visited on 05/11/2015) (cit. on pp. 12, 13, 15, 16, 52).

[33] *Weka wiki. How do I use the package manager?* URL: http://weka.wikispaces.com/How+do+I+use+the+package+manager? (visited on 05/13/2015) (cit. on pp. 12, 14).

[34] *Weka wiki. Pluggable evaluation metrics*. URL: http://weka.wikispaces.com/Pluggable+evaluation+metrics (visited on 05/11/2015) (cit. on p. 12).

[35] *Weka wiki. Related Projects list*. URL: http://weka.wikispaces.com/Related+Projects (visited on 04/27/2015) (cit. on p. 9).

[36] *Weka wiki. Unofficial packages for Weka 3.7*. URL: http://weka.wikispaces.com/Unofficial+packages+for+WEKA+3.7 (visited on 08/31/2015) (cit. on p. 9).

[37] *Wekalist. Visualization plugin, access to the result list names*. URL: http://list.waikato.ac.nz/pipermail/wekalist/2014-May/061041.html (visited on 06/01/2015) (cit. on p. 14).

[38] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005 (cit. on p. 9).

[39] Raymond W. Yeung. "A new outlook on Shannon's information measures." In: *IEEE Transactions on Information Theory* 37.3 (1991), pp. 466–474 (cit. on p. 4).